Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions

Department of Computer Science, University of Helsinki
2021

Balyo , T , Froleyks , N , Heule , M , Iser , M , Järvisalo , M & Suda , M (eds) 2021 , Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions . Department of Computer Science Report Series B , vol. B-2021-1 , Department of Computer Science, University of Helsinki , Helsinki .

http://hdl.handle.net/10138/333647

unspecified
publishedVersion

Proceedings of

SAT COMPETITION 2021

Solver and Benchmark Descriptions

Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda
(editors)

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2021-1

Helsinki 2021

PREFACE

The area of Boolean satisfiability (SAT) solving keeps on making progress. Besides new algorithms and better heuristics, refined implementation techniques turned out to be vital for the success story of SAT solving. To keep up the driving force in improving SAT solvers, SAT solver competitions provide opportunities for solver developers to present their work to a broader audience and to objectively compare the performance of their own solvers with that of other state-of-the-art solvers.

SAT Competition 2021 (SC 2021, `https://satcompetition.github.io/2021/`), a competitive event for SAT solvers, was organized as a satellite event of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT 2021). SC 2021 stands in the tradition of the previously organized main competitive events for SAT solvers: the SAT Competitions held 2002-2005, biannually during 2007-2013, 2014, 2016–2018, and 2020; the SAT Races held in 2006, 2008, 2010, 2015, and 2019; and SAT Challenge 2012.

SC 2021 consisted of a total of four tracks: Main Track (with CaDiCaL Hack, Crypto and No Limits sub-tracks), Incremental Library Track, Parallel Track, and Cloud Track. The crypto sub-track represents a second instantiation of a more domain-specific track in the SAT competitions, complementing the otherwise general tracks.

There were two ways of contributing to SC 2021: by submitting one or more solvers to participate in the competition and by submitting interesting benchmark instances on which the submitted solvers could be evaluated in the competition. The rules of SC 2021 required all contributors to submit a short, 1-2 page long description as part of their contribution. This book contains these non-peer-reviewed descriptions in a single volume, providing a way of consistently citing the individual descriptions and finding out more details on the individual solvers and benchmarks.

Successfully running SC 2021 would not have been possible without active support from the community at large. We would like to thank the StarExec initiative (`http://www.starexec.org`) for the computing resources needed to run SC 2021. Many thanks go to Aaron Stump for his invaluable help in setting up StarExec to accommodate for the competition's needs. Furthermore, we thank Amazon for providing the resources and support to develop parallel and distributed solvers on the AWS cloud and for executing the Cloud and Parallel tracks. Finally, we thank CAS Software Karlsruhe for sponsoring the awards.

Finally, we would like to emphasize that a competition does not exist without participants: we thank all those who contributed to SC 2021 by submitting either solvers or benchmarks and the related description.

Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, & Martin Suda
SAT Competition 2021 Organizers

Contents

Benchmark Descriptions

SOLVER DESCRIPTIONS

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

CaDiCaL, Kissat, Paracooba Entering the SAT Competition 2021

Armin Biere Mathias Fleury Maximilian Heisinger
Institute for Formal Models and Verification
Johannes Kepler University Linz

***Abstract*—This system description describes updates to our sequential SAT solvers CaDiCaL and Kissat submitted to the main track as well as updates to our distributed cube-and-conquer solver Paracooba submitted to the cloud track.**

CaDiCaL 1.4.0

The competition organizers decided to use CaDiCaL as basis for a "hack track". Version 1.4.0 of CaDiCaL used in this track differs from the version submitted to the SAT Competition 2020 [1] as follows.

First, our version of "reason side bumping" [2] not only bumps literals in reason clauses of literals in the learned clause, but, depending on a run-time recursion depth parameter, also bumps reason literals of literals in reason clauses recursively. By default the recursion depth limit was 1, which lead to the same behaviour as [2]. Now, in "stable mode", focusing on satisfiable instances with few restarts and smoothed bumping [3], we have increased this recursion depth limit to 2.

Second, to compute several statistics, CaDiCaL uses exponential moving averages, particularly for controlling restarts [4], [5]. Initializing these averages is non-trivial and actually leads to biased estimates. For instance, without proper initialization, the slow moving average of the LBD (glucose levels) of learned clauses ramps up too slowly, trailing the fast moving average, which in turn triggers unmotivated restarts initially. We proposed a partial solution in [5] and implemented another improvement based on over-approximating the smoothing factor geometrically. With CaDiCaL 1.4.0 we adopted the method for initializing exponential moving averages proposed in the ADAM approach [6], which maintains and uses a correction factor to obtain an unbiased average.

Finally, and most importantly, the version of CaDiCaL submitted to the SAT Competition 2020 unfortunately failed to export the assignment found in local search minimizing the number of falsified clauses back to the CDCL loop as saved phases (due to a change in semantics of `copy_phases`), which in essence rendered the local search component completely useless. And indeed, our post-competition experiments showed, that this "heuristic bug" resulted in solving fewer instances during the competition. In version 1.4.0 saved phases are again explicitly overwritten at the end of local search with the minimum assignment found during local search.

Supported by Austrian Science Fund (FWF) projects W1255-N23 and S11408-N23, by the LIT AI and LIT Secure and Correct Systems Labs and the LIT project LOGTECHEDU all three funded by the State of Upper Austria.

CaDiCaL SC2021

On top of version 1.4.0 of CaDiCaL, as used in the "CaDiCaL hack track", we have added a light version of (what we call) "shrinking" by Feng & Bacchus presented at SAT 2020. Like the original version [7], our version [8] shortens learned clauses by calculating the unique implication point (UIP) on every level of the conflict clause, restricted to not adding literals on lower levels. Unlike Feng & Bacchus's version, our algorithm is efficient enough to be executed unconditionally and minimizes the clause at the same time. This technique is particularly effective when long clauses are learned as in the planning track of the SAT Competition 2020. For more details please refer to [8].

Kissat SC2021

We have also implemented "shrinking" [8] in Kissat, which beside reducing the length of learned clauses yields the same run-time improvements on instances from the planning track, without degrading performance on other instances, even though the percentage of time spent in conflict analysis (including clause minimization and shrinking) goes up.

The local search procedure either imports decision phases from the CDCL solver or in an alternating fashion uses previously best assignments computed by local search, similar in spirit to the "cache" component in YalSAT [9]. This allows the local search to continue where it left off with the best assignments it found earlier. In addition of using a fixed CB value of 2.0 we also allow to interpolate it based on average clause length and further use three variants of fixed clause weights. These "strategies" are changed at each call to the local search procedure (as during "restarts" in YalSAT).

Beside using the new method described above to initialize exponential moving averages the new version of Kissat

- reduced the number of rephasing methods by removing the random and the flipped rephasing,
- computes backbones on the binary implication graph,
- schedules variable elimination without a priority queue,
- bounds reason side variable bumping, and

uses its default non-compact memory configuration for the competition in order to allow the solver to go beyond 24 GB main memory (as the organizers now announced to use 128 GB main memory during the competition).

Finally, we use a new method for semantic gate extraction in bounded variable elimination of a fixed candidate variable x.

Let $F = F_x \wedge F_{\bar{x}} \wedge F'$, where F_ℓ is the CNF of clauses of F which contain literal ℓ, with $\ell \in \{x, \bar{x}\}$, and where $(F_x \wedge F_{\bar{x}})$ is called the *environment* of x. The remaining clauses of F without x nor $\bar{x}$ are collected in F'. Given CNFs H_x, $H_{\bar{x}}$ where clauses in H_ℓ all contain ℓ, we define the set of resolvents

$$H_x \otimes H_{\bar{x}} = \{(C \vee D) \mid (C \vee x) \in H_x,\ (D \vee \bar{x}) \in H_{\bar{x}} \text{ and } (C \vee D) \text{ not trivial}\}.$$

As usual we interpret a CNF also as a set of clauses. Our goal is to eliminate x through resolution, that is replacing F by $(F_x \otimes F_{\bar{x}}) \wedge F'$.

Already in [10], which introduced the SATELITE preprocessor, it was proposed to extract subsets of "gate clauses" from F_x and $F_{\bar{x}}$ which encode "circuit gates" with output x, also called definitions of x. Resolving these gate clauses against each other results in tautological (trivial) resolvents, and, in particular, this situation allows to ignore resolvents between non-gate clauses (since those are implied).

Finding such gate clauses was originally based on syntactic pattern matching, in essence inverting the Tseitin encoding. For details and a semantic variant inspired by BDD algorithms and implemented in Lingeling [11] see [12]. In KISSAT we follow more recent semantic approaches with applications in model counting [13] and QBF reasoning [14], which use a SAT solver as oracle to find gate clauses.

Let x be a candidate variable, which is tried to be eliminated without increasing the number of clauses much, and, for which all necessary resolvents have to be generated. If syntactic pattern matching for a Tseitin encoding of an AND, XOR or IF-THEN-ELSE gate with x as output fails, then our new version of KISSAT tries to extract gate clauses semantically by checking satisfiability of $(F_x|_{\bar{x}}) \wedge (F_{\bar{x}}|_x)$, i.e., the formula which is obtained by removing the occurrences of x in F_x and of $\bar{x}$ in $F_{\bar{x}}$ and then conjoining the result. If this formula is unsatisfiable we compute a clausal core which in turn can be mapped back to original gate clauses G_x and $G_{\bar{x}}$ in the environment (by adding back x resp. $\bar{x}$). Let H_ℓ be the remaining clauses of F_ℓ with $F_\ell = G_\ell \wedge H_\ell$. Then it turns out that $F_x \otimes F_{\bar{x}}$ can be reduced to $(G_x \otimes G_{\bar{x}}) \wedge (G_x \otimes H_{\bar{x}}) \wedge (G_{\bar{x}} \otimes H_x)$ and thus $(H_x \otimes H_{\bar{x}})$ can be omitted.[1] The effect is that fewer resolvents are generated and thus more variables can be eliminated.

To see that the last formula can be omitted assume that $A \wedge B$ is unsatisfiable and thus $\bar{A} \vee \bar{B}$ is valid. Therefore

$$(A \vee C) \wedge (B \vee D) \equiv (A \vee C) \wedge (B \vee D) \wedge (\bar{A} \vee \bar{B}) \Rightarrow (C \vee D)$$

using in essence two resolution steps for the implication. Setting $(A, B, C, D) = (G_x, G_{\bar{x}}, H_{\bar{x}}, H_x)$ shows the rest.

KITTEN

In order to check satisfiability and compute clausal cores of these co-factors of the environment of a variable we have implemented a simple sub-solver KITTEN with in-memory proof tracing and fast allocation and deallocation. If the conjunction of the co-factors of the environment are unsatisfiable we reduce through the API in KITTEN its formula to the clausal core, shuffle clauses and run KITTEN a second time which usually results in a smaller core and thus fewer gate clauses (increasing chances that the variable is eliminated).

If only one co-factor contains core clauses, we derive a unit clause instead. In this case the learned clauses in KITTEN are traversed to produce a DRAT proof trace sequence for this unit. This is one benefit of using a proof tracing sub-solver in contrast to the BDD inspired approach in Lingeling [11], which can not produce DRAT proofs easily. This KITTEN feature of extracting proofs in memory is also essential to produce proofs for "SAT sweeping" discussed next.

KISSAT SC2021 SWEEP

As in the SAT Competition 2020 we submitted the default configuration "KISSAT SC2021 DEFAULT" as well as "KISSAT SC2021 SAT". The latter uses target phases also during focused mode [15] and usually works better for satisfiable instances. Instead of submitting a configuration specialized for unsatisfiable instances to the SAT Competition 2021, we decided to submit some work in progress, which in principle we expect to also work better on unsatisfiable instances.

Using KITTEN as sub-solver we perform semantically complete "SAT sweeping" of small *extended environments* around each variable, which works as follows. For each candidate variable we allocate a fresh instance of KITTEN and traverse in breadth first search (BFS) the variable incidence graph (in which two variables share an edge if they occur in the same clause) and copy all clauses up to a certain "depth" limit away (the number of BFS generations) from the candidate variable. We start with the default depth limit of 2 and also limit the total number of copied clauses (1000) and variables (100).

After copying the environment clauses, we let KITTEN compute a satisfiable assignment of the extended environment. Note that an unsatisfiable environment actually results in the whole formula to be unsatisfiable. From this satisfying assignment we produce a candidate list of backbone variables and a partition of equivalent literal candidates. The accumulated time spent in KITTEN is further limited by "ticks" as in KISSAT [15].

Then for each backbone candidate, we assume its negation and call the sub-solver again. If the result is unsatisfiable we learn the unit (and optionally extract a DRAT proof trace). Otherwise we use the satisfying assignment provided by KITTEN to refine both the backbone candidate list and the partition of equivalent literal candidates. By randomizing and every third call flipping the saved phases before calling the sub-solver the number of necessary calls is reduced substantially.

In the last part we then try to prove for each pair of remaining equivalent literal pairs, whether they imply each other, through two sub-solver calls with corresponding assumptions. If both calls are unsatisfiable, the two literals are equivalent and are merged in a global union-find data structures (and again optionally a DRAT proof sequence is extracted). This union-find data structure is consulted during the copy phase to add additional clauses of equivalent literals (as well as the

[1] Resolvents among gate clauses are not necessarily tautological though.

equivalence). A failing satisfiable call refines the equivalent literal candidate partition and the process continues until no more equivalent literal candidates are left.

The advantage of this approach is that the effort is heavily bounded, i.e., propagation over a large number of variables is avoided and solving is completely decoupled from the main solver. This version of KISSAT is still considered work in progress and for instance lacks better (re)scheduling of candidate variables. For more details and references on "SAT sweeping" see [16], which tries to achieve the same effect, but uses global blocked clause decomposition. A more similar approach but without using a dedicated sub-solver was implemented in our discontinued SAT solver SPLATZ [17].

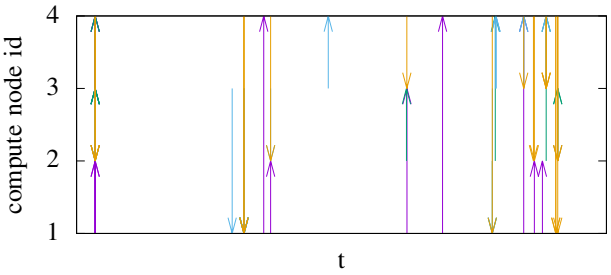
PARACOOBA SC2021

A new version of our solver PARACOOBA [15], [18] has been submitted to the cloud track. It is a distributed cube-and-conquer solver. The input DIMACS is analyzed and split once on the main node into multiple subproblem-branches. Each branch produces a tree of subproblems (cubes) that can be worked on independently. If branches of a cube-tree finish early, the branch is converted into a clause, which is distributed to all other solvers in the cluster. The first cube-tree is always favored when deciding on what task to work on next, so parallel cube trees do not starve the first tree-instance of available executors. Problems are re-split in case the CDCL-solver runs into a time-out depending on other solving times. Re-splits are done using the lookahead mechanism provided by CADICAL, which is also used as incremental solver to work on all generated cubes. To guard against problems that are faster to solve using a pure CDCL-based solving approach, KISSAT is running in parallel to the cube-and-conquer approach on the main node.

The architecture has been revised since last year, so that no ticks or delays are required anymore before tasks can be offloaded. Changes in a compute node's state are analyzed, compared to the last known received status of each known peer and sent only when the information gain of the new status is significant enough to warrant the transmission. This way, scheduling is more dynamic, relying less on overcommitting work (which lead to overfull queues) and instead offloading only when other nodes are nearly out of work. Furthermore, physical network connections no longer coincide with logical connections, which enables reconnects to work without losing previously sent messages. Timeouts are specified per peer and enforced with an improved keep-alive system that sends small messages if a connection has been idle for too long. This makes solving more resilient against network-based issues, even on commodity hardware. The latter also enables main nodes that only offload work and do not have workers themselves, which is useful for low-powered devices that only have wireless connections. Due to the low network bandwidth requirements, the connected peers can also be in the cloud and connected via SSH tunnels.

In order to test the improved offloading, the tracing tool DISTRAC was developed. This tool generates compact binary trace files that are concatenated after a run. A trace contains all required metadata (names, descriptions, causal dependencies between events) and can be analyzed using standard CLI tools, either in binary form, or after printing it in a textual column-based format. While analyzing the trace, events are sorted in order of system-wide occurrence, keeping causal relations intact. This enables easier debugging of network or solver events compared to merging log files, as filtering and selecting streams of data becomes easier.

One of the produced visualizations showing all offloads between compute nodes while solving a small benchmark can be seen below. Arrows are pointing from the compute node that currently works on a task to the new compute node that the task has been offloaded to. The Y-axis describes the compute node id, the X-axis the timestamp of an event. Highly utilized compute nodes offload their tasks to less utilized compute nodes, resulting in small clusters of arrows pointing to the same nodes, trying to maximize the active workers in the system without centralized coordination.



The solver has further been modularized into broker, communicator, solver, and runner components. These are either loaded at runtime as shared objects or statically linked into one binary. This mechanism enables using the PARACOOBA infrastructure for other problems, e.g., by implementing a custom solver module that uses the other mechanisms to automatically distribute tasks, or by changing the already provided solver to use different SAT-solvers. Automated unit- and system-tests work with dynamically loaded shared objects and can thus also be used to check third-party modules.

I. LICENSE

All our solvers are licensed under an MIT license and are available at https://github.com/arminbiere/cadical, http://fmv.jku.at/cadical for CADICAL, https://github.com/arminbiere/kissat, http://fmv.jku.at/kissat for KISSAT, and https://github.com/maximaximal/Paracooba for PARACOOBA.

REFERENCES

[1] A. Biere, "CaDiCaL at the SAT Race 2019," in *SAT Race 2019*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.

[2] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140.

[3] C. Oh, "Between SAT and UNSAT: the fundamental difference in CDCL SAT," in *SAT 2015*, ser. LNCS, M. Heule and S. A. Weaver, Eds., vol. 9340. Springer, 2015, pp. 307–323. [Online]. Available: https://doi.org/10.1007/978-3-319-24318-4_23

[4] G. Audemard and L. Simon, "Refining restarts strategies for SAT and UNSAT," in *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Milano, Ed., vol. 7514. Springer, 2012, pp. 118–126.

[5] A. Biere and A. Fröhlich, "Evaluating CDCL restart schemes," in *Proceedings of Pragmatics of SAT 2015, Austin, Texas, USA, September 23, 2015 / Pragmatics of SAT 2018, Oxford, UK, July 7, 2018*, ser. EPiC Series in Computing, D. L. Berre and M. Järvisalo, Eds., vol. 59. EasyChair, 2018, pp. 1–17.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[7] N. Feng and F. Bacchus, "Clause size reduction with all-UIP learning," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, ser. Lecture Notes in Computer Science, L. Pulina and M. Seidl, Eds., vol. 12178. Springer, 2020, pp. 28–45.

[8] M. Fleury and A. Biere, "Efficient all-uip learned clause minimization," 2021, submitted.

[9] A. Biere, "Yet another local search solver and Lingeling and friends entering the SAT Competition 2014," in *Proc. of SAT Competition 2014 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, A. Balint, A. Belov, M. Heule, and M. Järvisalo, Eds., vol. B-2014-2. University of Helsinki, 2014, pp. 39–40.

[10] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75.

[11] A. Biere, "Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010," Johannes Kepler University Linz, Tech. Rep., 2010.

[12] A. Biere, M. Järvisalo, and B. Kiesl, "Preprocessing in SAT solving," in *Handbook of Satisfiability*, 2nd ed., ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 391 – 435.

[13] J. Lagniez, E. Lonca, and P. Marquis, "Definability for model counting," *Artif. Intell.*, vol. 281, p. 103229, 2020.

[14] F. Slivovsky, "Interpolation-based semantic gate extraction and its applications to QBF preprocessing," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 12224. Springer, 2020, pp. 508–528.

[15] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[16] M. Heule and A. Biere, "Blocked clause decomposition," in *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. L. McMillan, A. Middeldorp, and A. Voronkov, Eds., vol. 8312. Springer, 2013, pp. 423–438.

[17] A. Biere, "Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016," in *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2016-1. University of Helsinki, 2016, pp. 44–45.

[18] M. Heisinger, M. Fleury, and A. Biere, "Distributed cube and conquer with paracooba," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, ser. Lecture Notes in Computer Science, L. Pulina and M. Seidl, Eds., vol. 12178. Springer, 2020, pp. 114–122.

[19] N. Feng and F. Bacchus, "Clause size reduction with all-UIP learning," in *SAT*, ser. LNCS, vol. 12178. Springer, 2020, pp. 28–45.

[20] R. Hickey, N. Feng, and F. Bacchus, "Cadical-trail, Cadical-alluip, Cadical-alluip-trail and Maple-LCM-Dist-alluip-trail at the SAT Competition," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, p. 10.

[21] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatiorial design," *IEEE Trans. Software Eng.*, vol. 23, no. 7, pp. 437–444, 1997. [Online]. Available: https://doi.org/10.1109/32.605761

[22] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 614–624. [Online]. Available: https://doi.org/10.1145/2970276.2970335

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Four CDCL Solvers Based On Learnt Clause Management And Restarts

1st Shunyang Bi, 1st Zhang Qu, 5th Hailong You
School of Microelectronics
XiDian University
Xi'an China
shybi@stu.xidian.edu.cn,
quzhang2019@stu.xidian.edu.cn,
hlyou@mail.xidian.edu.cn

2nd Meihua Liu
School of Electronic and Computer Engineering
Peking University Shenzhen Graduate School
Shenzhen Guangdong China
liumh@pku.edu.cn

3rd Pengfei Li, 4th Yang Zhang
EDA Group
SMIT Holdings Limited
Shenzhen Guangdong China
379401663@qq.com,
yanzhang@smit.com.cn

***Abstract*—this description introduce four CDCL solvers: Relaxed_LCFTP, Relaxed_LCFTP_V2, Relaxed_LCFTP_V3 and Relaxed_LCMCBDL_BLB, which are entering to the SAT Competition 2021.**

I. Introduction

This description presents four improved solvers based on Relaxed_LCMDCBDL_newTech (First SAT solver in SAT Competition 2020). The main improvements we made are involved the following two aspects: firstly, we changed the bumping evaluation method of learnt clauses; secondly, we adjusted the luby restart policy based on the variation tendency of the backtrack level during the solving process.

II. Two improvements

A. Learnt clause feedback to propation

In 2015, Chanseok Oh added a mid-tier of learnt clause database in COMiniSatPS, it quickly have become the dominant clause management strategy in recent SAT Race. The reason why this strategy so efficient is clearly clarified in the paper [1]. However, we tried to figure out the inner reason about it. And we found another decisive factor push this strategy to its solving limits of performance. The factor we observed that really matters in this principle is the feedback of learnt clause, which connects the whole solver components and guides the future search of algorithm. As such, the quality of learnt clause is the cornerstone of the solver [2]. From this respects, we guess the feedback of learnt clause have something with its quality. And we traced the call of learnt clause in the whole solver running except the conflict analyzing, which was the dominant strategy in recent year. Consequently, we found the important call of learnt clause in Boolean constrain propagation. In this process, with the assignment of literals, the algorithm search the related clauses and literals [3]. It would be high quality if a learnt clause is used repeatedly in propagation. Based on this , we tried to change the method of bumping activities, which directly represents the quality of learnt clauses.

Additionally, thanks to the professor Cai, and his excellent solver, Relaxed_LCMDCBDL_newTech, our details of this new strategy are implied in Relaxed_LCFTP. And the other version, Relaxed_LCFTP_V2 and his friend Relaxed_LCFTP_V3, are the parameter adjusting versions with our personal experiment.

B. Backtracking level-based optimization method for restarts

Relaxed_LCMDBDL_BLB used a restart policy based on the variation tendency of the backtracking level. The original intention of this improvement was whether there is a certain attribute does not change during the solving process when SAT solvers equipped with the same branch heuristics, preprocessing, and learning clause management but with different restart policies. We extracted the backtracking levels of the solution process and observe their properties in different restart policies, and found that the trend of the backtracking levels in the same instance was broadly similar. On the other hand, we configured the luby restart policy for Minisat to observe the performance of the UNSAT and SAT instance backtracking levels. We found an upward trend in the backtracking levels in SAT instances, and the trend is made more pronounced by gradually increasing the luby interval. And it is shown by experiments that the strategy of gradually increasing the luby interval is faster than the original version for SAT instances. Therefore, we use the same method to calculate the backtracking level as we used to calculate lbd, using the backtracking level as another threshold parameter for the restart.

References

[1] C. Oh, "Between SAT and UNSAT: The fundamental difference in CDCL SAT," in SAT, 2015.

[2] G. Audemard and L. Simon, "On the glucose sat solver," vol. 27, no. 01.

[3] M. Luo, C. Li, F. Xiao, F. Many`a, and Z. L¨u, "An effective learnt clause minimization approach for CDCL SAT solvers," in Proceedings of IJCAI 2017, 2017, pp. 703–711.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Kissat_MAB: Combining VSIDS and CHB through Multi-Armed Bandit

Mohamed Sami Cherif, Djamal Habet and Cyril Terrioux
Aix-Marseille Université, Université de Toulon, CNRS, LIS, Marseille, France
{mohamed-sami.cherif, djamal.habet, cyril.terrioux}@univ-amu.fr

***Abstract*—This document describes the Kissat_MAB solver which is based on the solver Kissat, winner of 2020 SAT competition. We augmented Kissat with a Multi-Armed Bandit (MAB) framework which combines the Variable State Independent Decaying Sum (VSIDS) and the Conflict-History Based (CHB) branching heuristics by adaptively choosing a relevant heuristic at each restart using the Upper Confidence Bound (UCB) strategy.**

***Index Terms*—SAT solver, Heuristics, Multi-Armed Bandit**

I. Introduction

Conflict Driven Clause Learning (CDCL) [8] solvers are known to be efficient on structured instances and manage to solve ones with a large number of variables and clauses. An important component in such solvers is the branching heuristic which picks the next variable to branch on. The Variable State Independent Decaying Sum (VSIDS) [9] has been the dominant heuristic since its introduction two decades ago. Recently, Liang and al. devised a new heuristic for SAT, called Conflict History-Based (CHB) branching heuristic [6], and showed that it is competitive with VSIDS. In the last years, VSIDS and CHB have dominated the heuristics landscape as practically all the CDCL solvers presented in recent SAT competitions and races incorporate a variant of one of them.

Recent research has shown the interest of machine learning in designing efficient search heuristics for SAT [5]–[7] as well as for other decision problems [4], [10]–[12]. One of the main challenges is defining a heuristic which can have high performance on any considered instance. Indeed, a heuristic can perform very well on a family of instances while failing drastically on another. To this end, we use reinforcement learning under the Multi-Armed Bandit (MAB) framework to pick an adequate heuristic among CHB and VSIDS for each instance. The MAB takes advantage of the restart mechanism in modern CDCL algorithms to evaluate each heuristic and choose the best one accordingly. The MAB uses the Upper Confidence Bounds (UCB) [1] strategy to select an arm at each restart.

II. Combining VSIDS and CHB through MAB

Let $A = \{VSIDS, CHB\}$ be the set of arms for the MAB containing different candidate heuristics. The proposed framework selects a heuristic $a \in A$ at each restart of the backtracking algorithm according to the Upper Confidence Bound (UCB) [1] policy. To choose an arm, UCB relies on a reward function calculated during each run to estimate the performance of the chosen branching heuristic. We choose a reward function that estimates the ability of a heuristic to reach conflicts quickly and efficiently. If t denotes the current run, the reward of arm $a \in A$ is calculated as follows:

$$r_t(a) = \frac{log_2(decisions_t)}{decidedVars_t}$$

$decisions_t$ and $decidedVars_t$ respectively denote the number of decisions and the number of decision variables, i.e. variables which were branched on at least once, in the run t. This reward function is adapted from the explored sub-tree measure introduced in [10].

The UCB1 algorithm [1] is used to select the next branching heuristic within the set of candidate heuristics A. The following parameters are maintained for each candidate arm $a \in A$:

- $n_t(a)$ is the number of times the arm a is selected during the t runs,
- $\widehat{r}_t(a)$ is the empirical mean of the rewards of arm a over the t runs.

UCB1 thus selects the arm $a \in A$ that maximizes $UCB(a)$ which is defined as follows :

$$UCB(a) = \widehat{r}_t(a) + c.\sqrt{\frac{ln(t)}{n_t(a)}}$$

The left-side term of $UCB(a)$ aims to put emphasis on arms that received the highest rewards. Conversely, the right-side term ensures the exploration of underused arms. The parameter c can help to appropriately balance the interchange between the exploitation and exploration phases in the MAB framework.

III. Implementation

We implement this idea in Kissat [3] which won first place in the main track of the SAT Competition 2020. Note that this solver is a condensed and improved reimplementation of the reference and competitive solver CaDiCaL [2], [3] in C. We maintain the VSIDS variant already implemented in Kissat which is similar to Chaff's where all analyzed variables are bumped after every conflict [9]. We also augment the solver with the heuristic CHB as specified in [6] except that we update the scores of the variables in the last decision level after unit propagation. In addition, we set the parameter c to 2. The rewards in UCB are initialized by launching each heuristic once during the first restarts. It is important to note that the only modified components of the solver are

the decision component and the restart component, i.e. all the other components as well as the default parameters of the solver are left untouched.

REFERENCES

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[2] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[3] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[4] Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. On the Refinement of Conflict History Search Through Multi-Armed Bandit. In Miltos Alamaniotis and Shimei Pan, editors, *Proceedings of 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 264–271. IEEE, 2020.

[5] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning. *CoRR*, 2019.

[6] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3434–3440, 2016.

[7] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140, 2016.

[8] João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[9] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, 2001.

[10] Anastasia Paparrizou and Hugues Wattez. Perturbing branching heuristics in constraint solving. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, pages 496–513, Cham, 2020. Springer International Publishing.

[11] Hugues Wattez, Frederic Koriche, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. In *Proceedings of the European Conference on Artificial Intelligence*, 2020.

[12] Wei Xia and Roland H. C. Yap. Learning Robust Search Strategies Using a Bandit-Based Approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 6657–6665, 2018.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Four CDCL solvers based on expLRB, expVSIDS and Glue Bumping

Md Solimul Chowdhury
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mdsolimu@ualberta.ca

Martin Müller
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mmueller@ualberta.ca

Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
jyou@ualberta.ca

***Abstract*—This document describes 4 CDCL SAT solvers: kissat_gb, kissat_crvr_gb, cms_expV_gbL and CaDiCaL_hack_gb which are entering the SAT Competition-2021. These solvers are based on three new ideas: 1) Guidance of Learning Rate Based (LRB) and Variable State Independent Decaying Sum (VSIDS) branching heuristics via random exploration amid pathological phases of conflict depression 2) Activity score bumping of variables which appear in the glue clauses, and 3) Common Reason decision Variable score Reduction (CRVR).**

I. Guidance of CDCL Branching Heuristics via Random Exploration during Conflict Depression

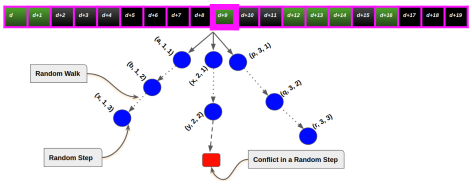
This approach is based on our observation that CDCL SAT solving entails clear non-random patterns of bursts of conflicts followed by longer phases of *conflict depression (CD)* [1]. During a CD phase a CDCL SAT solver is unable to generate conflicts for a consecutive number of decisions. To correct the course of such a search, we propose to use exploration to combat conflict depression. We therefore design a new SAT solver, called *expSAT*, which uses random walks in the context of CDCL SAT solving. In a conflict depression phase, random walks help identify more promising variables for branching. As a contrast, while exploration explores *future* search states, LRB and VSIDS relies on conflicts generated from the *past* search states.

A. expSAT algorithm

Given a CNF SAT formula $\mathcal{F}$, let $vars(\mathcal{F})$, $uVars(\mathcal{F})$ and $assign(\mathcal{F})$ denote the set of variables in $\mathcal{F}$, the set of currently unassigned variables in $\mathcal{F}$ and the current partial assignment, respectively. In addition to $\mathcal{F}$, *expSAT* also accepts four *exploration parameters* nW, lW, p_{exp} and ω, where $1 \leq nW, lW \leq uVars(\mathcal{F})$, $0 < p_{exp}, \omega \leq 1$. These parameters control the exploration aspects of *expSAT*. The details of these parameters are given below.

Given a CDCL SAT solver, *expSAT* modifies it as follows: (I) Before each branching decision, if a **substantially large CD phase** is detected then with probability p_{exp}, *expSAT* performs an *exploration episode*, consisting of a fixed number *nW* of random walks. Each walk consists of a limited number of *random steps*. Each such step consists of (a) the uniform random selection of a currently unassigned *step variable* and assigning a boolean value to it using a standard CDCL *polarity* heuristic, and (b) a followed by Unit Propagation (UP). A walk terminates either when a conflict occurs during UP, or after a fixed number *lW* of random steps have been taken. Figure 1 illustrates an exploration episode amid a CD phase. (II) In an exploration episode of nW walks of maximum length lW, the *exploration score expScore* of a decision variable v is the average of the *walk scores* $ws(v)$ of all those random walks within the same episode in which v was one of the randomly chosen decision variables. $ws(v)$ is computed as follows: (a) $ws(v) = 0$ if the walk ended without a conflict. (b) Otherwise, $ws(v) = \frac{\omega^d}{lbd(c)}$, with decay factor $0 < \omega \leq 1$, $lbd(c)$ the LBD score of the clause c learned for the current conflict, and $d \geq 0$ the *decision distance* between variable v and the conflict which ended the current walk: If v was assigned at some step j during the current walk, and the conflict occurred after step $j' \geq j$, then $d = j' - j$. We assign credit to all the step variables in a walk that ends with a conflict and give higher credit to variables closer to the conflict. (III) The novel branching heuristic `expVSIDS` adds VSDIS score and *expScore* of the variables that participated in the most recent exploration episode. For `expVSIDS`, a variable v^* with maximum combined score is selected for branching. (IV) All other components remain the same as in the underlying CDCL SAT solver.

Fig. 1: The 20 adjacent cells denote 20 consecutive decisions starting from the d^{th} decision, with $d > 0$, where a green cell denotes a decision with conflicts and a black cell denotes a decision without conflicts. Say that amid a CD phase, just before taking the $(d+9)^{th}$ decision, *expSAT* performs an exploration episode via 3 random walks each limited to 3 steps . The second walk ends after 2 steps, due to a conflict. A triplet (v, i, j) represents that the variable v is randomly chosen at the j^{th} step of the i^{th} walk.



II. Glue Variable Bumping

Let a CDCL SAT solver M is running a given SAT instance $\mathcal{F}$ and the current state of the search is S. We call the variables that appeared in at least one glue clause up to the current state S *Glue Variables*. We design a structure-aware variable score bumping method named *Glue Bumping* (GB) [2], based on the notion of *glue centrality (gc)* of glue variables. Given a glue variable v_g, glue centrality of v_g dynamically measures the fraction of the glue clauses in which v_g appears, until the current state of the search. Mathematically, the glue centrality of v_g, $gc(v_g)$ is defined as follows:

$$gc(v_g) \leftarrow \frac{gl(v_g)}{ng}$$

, where ng is the total number of glue clauses generated by the search so far. $gl(v_g)$ is the glue level of v_g, a count of glue clauses in which v_g appears, with $gl(v_g) \leq ng$.

A. The GB Method

The GB method modifies a CDCL SAT solver M by adding two procedures to it, named *Increase Glue Level* and *Bump Glue Variable*, which are called at different states of the search. We denote by M^{gb} the GB extension of the solver M.

Increase Glue Level: Whenever M^{gb} learns a new glue clause g, before making an assignment with the first UIP variable that appears in g, it invokes this procedure. For each variable v_g in g, its glue level, $gl(v_g)$ is increased by 1.

Bump Glue Variable: This procedure bumps a glue variable v_g, which has just been unassigned by backtracking. First a bumping *factor (bf)* is computed as follows:

$$bf \leftarrow activity(v_g) * gc(v_g)$$

, where $activity(v_g)$ is the current activity score of the variable v_g and $gc(v_g)$ is the glue centrality of v_g. Finally, the activity score of v_g, $activity(v_g)$ is bumped as follows:

$$activity(v_g) \leftarrow activity(v_g) + bf$$

III. Common Reason decision Variable score Reduction (CRVR)

During a CDCL search, a single decision can generate more than one conflicts, from which multiple clauses are learned. We refer decisions with multiple conflicts as `mc` decisions. Let a `mc` decision $\mathcal{M}$ generates n conflicts, from which it learns a sequence of clauses $\mathcal{L} = (L_1 \dots L_n)$. For any clause $L_i \in \mathcal{L}$, let $L_i = R_i \vee \neg f$, where R_i be the reason clause and f be the unique implication point for the conflict that generates L_i.

We call the set of decision variables in $R_1 \cap \dots \cap R_i \cap \dots \cap R_n$ as Common Decision Variables (CRVs) for $\mathcal{M}$. CRVs are the common decision variables over the reason clauses in $\mathcal{M}$.

The CRVR scheme decreases the activity scores of those CRVs, which correspond to `mc` decisions with average LBD score higher (i.e., have lower quality learned clauses) than the recent search average.

IV. Solvers Description

We have submitted four CDCL SAT solvers to SAT Competition-2021, which are based on four combinations of the three approaches described in the previous sections. Our solvers are implemented on top of the solver kissat_sat (kissat with `sat` configuration) [3], CaDiCaL1.4.0 (base solver for the CaDiCaL hack track) [4], and CryptoMiniSAT5.8.0 [5]. In the following, we describe our solvers:

a) ***kissat_gb****:* This solver implements the GB method on top of kissat_sat. kissat_sat employs two branching heuristics: VSIDS and VMTF. In kissat_gb, the GB scheme is kept active only when VSIDS is active.

b) ***kissat_crvr_gb****:* This solver implements the GB and CRVR method on top of kissat_sat. In kissat_crvr_gb, the GB and CRVR schemes are kept active only when VSIDS is active.

c) ***cms_expV_gbL****:* The baseline CryptoMiniSAT5.8.0 employs a combination of three branching heuristics: LRB, VSIDS and VMTF. This system extends this baseline by implementing the GB method on top of LRB, and by replacing VSIDS with `expVSIDS`.

d) ***CaDiCaL_hack_gb****:* This system implements the GB method on the top of CaDiCal1.4.0, only when VSIDS is active in the baseline system. CaDiCaL_hack_gb is submitted to the CaDiCaL hack track of the SAT compeition-2021.

References

[1] Md Solimul Chowdhury and Martin Müller and Jia You, Guiding CDCL SAT Search via Random Exploration amid Conflict Depression, To appear in Proceedings of AAAI-2020.

[2] Md. Solimul Chowdhury, Martin Müller, Jia-Huai You, Exploiting Glue Clauses to Design Effective CDCL Branching Heuristics. In Proceedings of CP 2019: 126-143.

[3] Armin Biere Katalin Fazekas Mathias Fleury Maximilian Heisinge. CADICAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING Entering the SAT Competition 2020, In Proceedings of SAT Competition 2020:50-52.

[4] CaDiCal 1.4, https://github.com/arminbiere/cadical/tree/rel-1.4.0, access date: 02-April-2021.

[5] CryptoMiniSat 5.8.0, https://github.com/msoos/cryptominisat/releases, access date: 02-April-2021.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Cadical_SCAVEL and its friends at the SAT Competition 2021

1st Zhihui Li, 2nd Guanfeng Wu, 3rd Yang Xu, 4th Huimin Fu
School of Mathematics
National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University
Chengdu, China
lizhihui@swjtu.edu.cn, wgf1024@swjtu.edu.cn, xuyang@swjtu.edu.cn, fhm6688@my.swjtu.edu.cn

Abstract— this document describes six solvers from Scavel: Cadical_SCAVEL01/02, Relaxed_LCMDCBDL_SCAVE-L01/02, and two parallel solvers abcd_para18_Scavel and P-MCOMSPS-STR-32-SC at the SAT Competition 2021.

I. INTRODUCTION

The base solvers we used to implement our techniques are cadical-alluip-trail and Relaxed_LCMDCBDL_newTech, obtained from the SAT Competition 2020 [1]. Based on these two very competitive solvers, some minor changes mainly include the following technical solutions: active backtrack, randomization of 2-watched scheme[2],Highlight the role of the stochastic local search solver in the Inception phase, and the adjustment of the program flow with the previous centralized technology.

II. ALGORITHM AND IMPLEMENTATION DETAILS

A. Active Backtrack

When we look at the dynamic growth and backtracking /restart shortening of propagation sequences of a global perspective, we analyze the assignment queues propagated according to BCP that these sequences are initiated by both conflict-causing and non-conflict-causing decisions, at the same time, they are also initiated by continuous conflict-causing decisions and continuous non-conflict-causing decisions.

Based on the empirical analysis, we obtained the variation law of the general solution process of continuous conflict decision and continuous non-conflict decision, and carried out two improvements based on this representational quantity. According to the continuous non-conflict decision model, a new restart standard is determined, the concept of Active Backtrack is proposed, and an Active Backtrack module is added between conventional conflict backtracking and general quick restart. This is different from the existing Trail Saving [3].

B. Randomization of 2-watched Scheme

In a typical CDCL implementation, a data structure called 2-watched scheme is commonly used because the unit Propagation needs to detect unit clauses as efficiently as possible. In cases where more than a unit clause may be examined at the same time, the 2-watched scheme usually presents one of them in order. This order is formed by the clause literals order or the construction of Elements of watches by the previous propagation. Usually the order is fixed unless it is randomized periodically.

C. Highlight the role of the stochastic local search solver

As famous example of combining CDCL with SLS, CaDiCaL and Relaxed_LCMDCBDL use respectively the two stochastic local search (SLS) solvers (probSAT and ccnr) at a specific point in time to get the value of the argument close to the solution space. Usually the SLS solver is fast and efficient when solving hard small-scale problems. For a smaller sample （the number of variables is less than 3000）, We enhance the SLS solver solution time (the number of flips) in an attempt to obtain the result directly in the initial stage , rather than just providing the initial phase for all the variables.

D. External Restart Frame

Quick restart technology is very important, especially important to solving UNSAT incidents. As a major technical module of the CDCL framework, restart is triggered multiple times of the solver function. Before and after restart, the values of the variables score of the decision branch are not changed, and the 2-watched scheme usually presents one of them in order. This order is formed by the clause literals order or the construction of Elements of watches by the previous propagation. Usually the order for every literal 2-watches is fixed. We adjust the program flow with external restart frame to start the solution processes several times based on the number of restarts that have occurred by adding an outer loop around the solve function. Before the solve function is called, we increase the decision branches value of inactive variable, manage the size of three clauses set (core, Iter2 and local) boldly and randomize the existing 2-watched Scheme. They are mainly for eliminating the adverse effect of the former solution stage on the latter as far as possible.

III. SOLVERS

We add the learnt clause used frequency strategy[4] to other parallel solvers to see the effect of this strategy in other parallel solvers. We build our parallel Solvers based on P-MCOMSPS-STR-32[5] and abcdSATi20[6], So the name of parallel solvers are “ P-MCOMSPS-STR-32-SC” and “ abcd_para18_Scavel”.

The Cadical_SCAVEL01/02 solvers in this submission are small amount modifications of CaDiCaL [1] that participated in SAT competition 2020, which implement our techniques of II.C. Especially, Cadical_SCAVEL01 is based on CADICAL-SC2020, and Cadical_SCAVEL01 is based on adical-alluip-trail.

The Relaxed_LCMDCBDL_SCAVEL01/02 solvers in this submission are modifications of Relaxed_LCMDCBDL

_newTech [1] that participated in SAT competition 2020. Especially, Relaxed_LCMDCBDL_SCAVEL01 implements our techniques of II. A, and Relaxed_LCMDCBDL_SCA-VEL02 represents our techniques of II. A ~ II. A D.

IV. ACKNOWLEDGMENTS

The author would like to thank the developers of all predecessors of Minisat, MapleLCMDistChronoBT-DL, CaDiCaL, abcdsatptwenty, P-MCOMSPS-STR-32, Relaxed_LCMDCBDL_newTech and all the authors who contributed the modifications that have been integrated. Their solvers are the foundation of our learning and improvement. This work also supported by the Fundamental Research Funds for the Central Universities (No.2682020CX59).

REFERENCES

[1] Hickey R, Feng N, Bacchus F. Cadical-trail, Cadical-alluip, Cadical-alluip-trail, and Maple-LCM-Dist-alluip-trail at SAT Competition 2020[J]. SAT COMPETITION 2020, 10.

[2] M. Moskewicz, C. Conor, Y. Zhao, L. Zhang and S. Malik, Chaff: Engineering an efficient SAT solver, in Proc. DAC'01 (2001).

[3] Ramos A., van der Tak P., Heule M.J.H. (2011) Between Restarts and Backjumps. In: Sakallah K.A., Simon L. (eds) Theory and Applications of Satisfiability Testing - SAT 2011. SAT 2011. Lecture Notes in Computer Science, vol 6695. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-21581-0_18

[4] Wu, G., Chen, Q., Xu, Y., & He, X. (2018). A Hybrid Learnt Clause Evaluation Algorithm for SAT Problem. International Journal of Computational Intelligence Systems, 12(1), 250-258..

[5] Vallade, V., Le Frioux, L., Baarir, S., Sopena, J., & Kordon, F. P-MCOMSPS-STR: a Painless-based Portfolio of MapleCOMSPS with Clause Strengthening. SAT COMPETITION 2020, 56.

[6]Chen, J. Optsat, Abcdsat and Solvers Based on Simplified Data Structure and Hybrid Solving Strategies. SAT COMPETITION 2020, 25.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Maple_MBDR_Cent_PERM and Maple_MDBR_BJL in the 2021 SAT Solver Competition

Sima Jamali
Simon Fraser University
Vancouver, Canada
sja88@sfu.ca

David Mitchell
Simon Fraser University
Vancouver, Canada
mitchell@cs.sfu.ca

***Abstract*—We give a brief description of our solvers Maple_MBDR_Cent_PERM and Maple_MBDR_BJL. Maple_MBDR_Cent_PERM stores high-centrality learned clauses permanently. Maple_MBDR_BJL, uses a simple and cheap scheme to learn additional small clauses. Both solvers are based on Relaxed_Maple_LCMDCBDL_newtech, the second place solver from the main track of the 2020 SAT Solver Competition.**

***Index Terms*—High-Centrality Clauses, Permanent, Learning**

I. Introduction

Relaxed_Maple_LCMDCBDL_newtech won the silver medal of the main track of the 2020 SAT Solver Competition [3]. This is a recent member of the MapleSAT family of solvers that has been improved over the last 5 years [1], [2], [4], [6], [8], [9]. Our submissions modify the clause maintenance strategy of Relaxed_Maple_LCMDCBDL_newtech, which was inherited from COMiniSatPS and involves three stores of learnt clauses: Core, Tier2 and Local [5], [6]. Learned clauses are placed in one of these stores based on their LBD. Small LBD clauses are placed in Core and retained permanently.

Clause Centrality was introduced in [10] and shown to be a useful clause quality measure. Maple_MBDR_Cent_PERM places high-centrality clauses in Core regardless of their LBD. Small clauses are understood to be valuable, and many solvers store very small clauses permanently. Maple_MBDR_BJL has a simple scheme to learn an additional small clause after each backjump to a small decision level.

II. High-Centrality Permanent clauses

The centrality of a clause is the mean betweenness centrality of its variables. To compute centralities, we generate the primal graph of the input CNF (after pre-processing). The betweenness centrality of a vertex (variable) v is the number of shortest paths between pairs of vertices excluding v, that visit v. It is defined by $g(v) = \sum_{s \neq v \neq t}(\sigma_{s,t}(v)/\sigma_{s,t})$, where $\sigma_{s,t}$ is the number of shortest s-t paths and $\sigma_{s,t}(v)$ is the number of those passing through v, normalized to $[0, 1]$ [11].

This work was supported by the Natural Sciences and Engineering Council of Canada (NSERC) through a Discovery Grant to the second author.

We use Brandes algorithm [12] to compute centrality values. Generating the graph and computing centrality values are memory and time intensive for large formulas, so we compute centralities only for formulas with at most 100,000 clauses after pre-processing. We also limit the time for centrality computation to 150 seconds. We use the base solver without modification for formulas without centralities. For formulas with centralities, high centrality (HC) learned clauses (those with centrality greater than a threshold CT), are stored in Core, regardless of their LBD. We aim to include at least the 0.02% of learned clauses with highest centrality in Core. We set an initial threshold of $CT \geq 0.008$. Every $100,000$ conflicts, if the number of HC clauses in Core is less than 0.02% of all learned clauses, CT is reduced by 0.001, but it is never reduced below 0.004. We submitted two versions:

- Maple_MBDR_Cent_PERM_10K: This solver adds at most the first 10,000 HC clauses to Core.
- Maple_MBDR_Cent_PERM_75K: This solver adds at most the first 75,000K HC clauses to Core.

III. BackJump Learning

Standard conflict analysis schemes derive one clause, called the 1-UIP clause, at each conflict. To Maple_MBDR_BJL, we add a simple and inexpensive scheme to learn additional small clauses.

BackJump Learning Scheme (BJL): Assume a conflict at level x, meaning after assigning x literals $l_1, l_2, .., l_x$ to true, a conflict is reached. After conflict analysis the solver backjumps to a level b and learns a 1UIP clause $C = \{m_1, m_2, ..., m_{i-1}, m_i\}$. Only one literal m_i from C belongs to level x, and $b < x$, so after the first b decisions, if we had C in the clause database, unit propagation would prevent this conflict by assigning m_i true. Therefore, we can also learn clause $C_2 = \{\neg l_1, \neg l_2, ...\neg l_b, m_i\}$. For small values of b, this new learned clauses is small.

We submitted two versions:

- Maple_MBDR_BJL6_Local: This solver uses the BJL learning scheme described above and sets $b = 6$. The new learned clauses are stored in Local regardless of their LBD value.

- Maple_MBDR_BJL7_Tier2: This solver uses the BJL learning scheme described above and sets $b = 7$. The new learned clauses are stored in tier2 regardless of their LBD value.

REFERENCES

[1] A. Nadel and R. Vadim, "Chronological Backtracking," in Proceedings of SAT, 2018, pp. 111–121.

[2] A. Nadel and R. Vadim, "Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking," in Proceedings SAT Competition 2018 - Solver and Benchmark Descriptions, 2018, pp. 29.

[3] SAT Competition 2020, *https://satcompetition.github.io/2020/*

[4] C. Oh, "Between SAT and UNSAT: The Fundamental Difference in CDCL SAT," in Proceedings of SAT, 2015, pp. 307–323.

[5] C. Oh, "Improving SAT solvers by exploiting empirical characteristics of CDCL", Ph.D. dissertation, New York University, 2016.

[6] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in Proceedings of SAT, 2016, pp. 123–140.

[7] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in Proceedings of IJCAI, 2009, pp. 399–404.

[8] M. Luo, C.-M. Li, F. Xiao, F. Manya, and Z. Lu, "An effective learnt clause minimization approach for CDCL SAT solvers," in Proceedings of IJCAI, 2017, pp. 703–711.

[9] X. Zhang and Sh. Cai, "Relaxed Backtracking with Rephasing," in Proceedings of SAT Competition 2020 - Solver and Benchmark Descriptions, 2020, pp. 15-16.

[10] S. Jamali and D. Mitchell, "Centrality-based improvements to CDCL heuristics Authors," in Proceedings of SAT, 2018, pp. 122-131.

[11] L. Freeman, "A set of measures of centrality based on betweenness," in Journal of Sociometry, volume 40(1), 1977, pp. 35-41.

[12] U. Brandes, "A faster algorithm for betweenness centrality," in Journal of Mathematical Sociometry, volume 25(2), 2001, pp. 163–177.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Optsat, Abcdsat and Maple_simp : Speed up Solving Satisfiable Instances

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn

***Abstract*—In order to speed up solving satisfiable instanceswe use a simple rephasing technique to our previous SAT solvers such as abcdsat, maple_simp and optsat.**

I. Introduction

Decision variable phasing policies are very important in CDCL solvers. Here we use a simple phasing policy similar to the phasing policy of [1], [2] to improve the speed of our SAT solvers.

II. A Simplified Rephasing Technique

We use a technique similar to the rephasing technique in CaDiCaL [1] and Relaxed_LCMDCBD [2] to the speed of solvers on satisfiable instances. Unlike CaDiCaL and Relaxed_LCMDCBD, we delete the local search solver for rephasing. This technique is actually a extension of the bit-eoncoding phase selection strategy in [3]. The simplified rephasing technique used this year may be described as follows. When the CDCL search reaches a conflict, the trail stack has the partial assignment of variables. We apply the unit propagation of CDCL and the random flip to extend the partial assignment to a full assignment. We save three full assignments to handle the variable phase problem. When the trail stack reaches a maximal length, the extended full assignment is called max_trail assignment. We get the current partial assignment based on the trail stack every 1500 restarts, and extend it to the full assignment, which is called current full assignment. The current full assignment with minimized unsatisfied clauses is called the best assignment. We maintain always the update of max_trail assignment, current full assignment and best assignment. When selecting the phase of decision variables, we use max_trail assignment, current full assignment and best assignment with 30%, 30% and 10% probability, respectively. The probability that the other assignment such as all true, all false, random, reverse etc is used is 30%.

III. Maple_simp21

Maple_simp21 participates the main track. It is an improved version of Maple_simp20 [4]. The main difference between them is that Maple_simp21 adds the simplified rephasing technique mentioned above.

IV. Optsat *m21*

This solver is submitted to the main track. It is an improved version of Optsat *m20* [4]. Different from Optsat *m20*, Optsat *m21* eliminates the hyper binary resolution in-processing. Another difference is Optsat *m21* adds the simplified rephasing technique mentioned above.

V. Optsat *R21*

This solver is submitted to the main track. It is a recursive version of Optsat *m21* mentioned above. The difference between them is that Optsat *R21* adds in-processing, which includes subsumption and variable elimination. The in-processing here is implemented by calling recursively the pre-processing subroutine of Optsat *m21*.

VI. abcdsat *i21*

This solver is submitted to the incremental library track. It is an improved version of abcdsat *i20* [4]. Compared with abcdsat *i20*, abcdsat *i21* adjusts some parameters, and adds adds the simplified rephasing technique mentioned above.

VII. abcdsat *p21*

This solver is submitted to the parallel track. It is an improved version of abcdsat *p18* [5], rather than last yeras version abcdsat *p20* [4]. Abcdsat *p21* uses at most 64 threads. 61 out of 64 threads solve the subproblem $F \wedge p$, where p and F are a pivot and the original problem respectively. The other 3 threads solve either the original problem or the simplified problem. The other difference from abcdsat *p18* is that abcdsat *p21* contains the simplified rephasing technique mentioned above.

References

[1] Biere, A.: Cadical at the sat race 2019. SAT RACE 2019, p.8
[2] Zhang, X., Cai, S.: Relaxed Backtracking with Rephasing, Proceedings of SAT competition 2020, pp.15–16
[3] Chen J.: A bit-eoncoding phase selection strategy for satisfiability solvers, in Proceedings of Theory and Applicationsf Models of Computation (TAMC'2014), LNCS, vol. 8402, Chennai, India, 2014, pp. 158-167
[4] Chen, J.: Optsat, Abcdsat and Solvers Based on Simplified Data Structure and Hybrid Solving StrategiesProceedings of SAT competition 2020, pp. 25–26
[5] Chen J.: AbcdSAT and Glucose hack: Various Simplifications and Optimizations for CDCL SAT solvers, Proceedings of SAT Competition 2018, pp.10-12

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

CleanMaple

Benjamin Kaiser and Robert Clausecker
Zuse Institute Berlin
Berlin, Germany
{kaiser,clausecker}@zib.de

Abstract—This document describes the SAT Solver CleanMaple, which is a refactored version of the SAT Competition 2018 winner Maple_LCM_Dist_ChronoBT [1].

Index Terms—SAT, refactoring, CDCL

I. Overview

The complex nature of the CDCL algorithm and the necessity of high performance implementations encourages a tight coupling of most subroutines and data structures in the source code. However the basic ideas on which CDCL algorithms are based are simple when contrasted to their implementation in Maple_LCM_Dist_ChronoBT. Most parts of the solver Maple_LCM_Dist_ChronoBT are included in one huge monolithic class and many of its methods are themselves massive, having more than 50, 100 or even 150 lines of dense code, resulting in a single source file for this class of almost 2000 lines of code. This design choice as well as most code of the solver can be traced back to the solver Minisat [2] [3] from which Maple_LCM_Dist_ChronoBT evolved over a time span of more than ten years with ideas and contributions from many different authors, most notably by the authors of [4], [5], [6], [7], [8] and [9]. The rather complicated code base leads to a steep learning curve for researchers that wish to develop SAT Solvers based on this state-of-the-art solver.

II. Description

In CleanMaple the two main subroutines

- Unit Propagation and
- the heuristic-based Branching,

and the three main data structures

- the clause database containing all original and learned clauses,
- the variable database containing the three-valued truth-value with respect to the current assignment and the polarities of all variables and
- the implication graph, i.e. the trail, used for fast conflict analysis

have been decoupled from the actual class. This leads to a solver that is much easier to understand. Furthermore, due to the refactoring the size of the binary of the solver was reduced significantly.

Acknowledgment

We want to express our gratitude towards the organizers of the SAT Competition 2021 for making such an event possible. Additionally we like to thank Florian Schintke for his support and the IT and Data Services members of the Zuse Institute Berlin for providing the infrastructure and their fast help. Also we like to thank the authors of Maple_LCM_Dist_ChronoBT and everyone else contributing to this solver.

References

[1] Vadim Ryvchin and Alexander Nadel, "Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking" in Proceedings of SAT Competition, 2018 p.29
[2] Eén, Niklas and Sörensson, Niklas, "An Extensible SAT-solver" in Theory and Applications of Satisfiability Testing, 2004 p.502–518
[3] Sörensson, Niklas and Eén, Niklas, "Minisat v1.13-a SAT solver with conflict-clause minimization" in International Conference on Theory and Applications of Satisfiability Testing, 2005
[4] Audemard, Gilles and Simon, Laurent, "Predicting learnt clauses quality in modern SAT solvers" in Proceedings of the 21st International Jont Conference on Artifical Intelligence, 2009 p.399–404
[5] Chanseok Oh, "COMiniSatPS the Chandrasekhar Limit and GHackCOMSPS" in Proceedings of SAT Competition 2016, p.29–30
[6] Liang, Jia Hui and Ganesh, Vijay and Poupart, Pascal and Czarnecki, Krzysztof, "Learning Rate Based Branching Heuristic for SAT Solvers" in Theory and Applications of Satisfiability Testing – SAT 2016, 2016, p.123–140
[7] Mao Luo and Chu-Min Li and Fan Xiao and Felip Manyà and Zhipeng Lü, "An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers" in Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, 2017, p.703–711
[8] F. Xiao, M. Luo, C.-M. Li, F. Manya, and Z. Lü, "MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose-3.0+width in SAT Competition 2017," in Proceedings of SAT Competition, 2017 p.22
[9] Nadel, Alexander and Ryvchin, Vadim, "Chronological Backtracking" in Theory and Applications of Satisfiability Testing – SAT 2018, 2018 p.111–121

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

CleanMaple_PriPro, CaDiCaL_PriPro and CaDiCaL_PriPro_no_bin

Benjamin Kaiser and Robert Clausecker
Zuse Institute Berlin
Berlin, Germany
{kaiser,clausecker}@zib.de

Abstract—Our experimental results suggest that some methods of rearranging the order in which clauses are propagated increase the performance in CDCL-solvers. CleanMaple_PriPro, CaDiCaL_PriPro and CaDiCaL_PriPro_no_bin are alterations of state-of-the-art SAT-solvers in which a novel approach of propagating some clauses with a severe priority increases their performance.

Index Terms—SAT, CDCL, propagation order

I. Priority Propagation

In all three solvers a second two-watch-literal-scheme of locally watched clauses is introduced. Newly learned conflict clauses are not registered to the standard two-watch-literal-scheme, but instead locally watched. Similarly, during conflict analysis each conflicting clause with an LBD of less than 7 not yet locally watched is deregistered from the standard two-watch-literal-scheme and instead registered to be locally watched. During propagation at each level the implications from all locally watched clauses at all levels are computed similarly to how binary clauses are propagated first in the SAT competition 2018 winner Maple_LCM_Dist_ChronoBT [1]. Every 10k conflicts all locally watched clauses are downgraded, i. e. deregistered from the second two-watch-literal-scheme and re-registered in the standard two-watch-literal-scheme.

II. Description of the solvers

The solver CleanMaple_PriPro is based on CleanMaple [2], which itself is based on Maple_LCM_Dist_ChronoBT. The solvers CaDiCaL_PriPro and CaDiCaL_PriPro_no_bin are based on CaDiCal [3]. In CleanMaple_PriPro binary clauses are never locally watched, but instead propagated immediately after locally watched clauses. This is due to the fact that binary clauses are watched in a separate watch-list in Maple_LCM_Dist_ChronoBT and have already been propagated with increased priority before. The solvers CaDiCaL_PriPro and CaDiCaL_PriPro_no_bin differ only by the fact whether binary clauses are considered for being locally watched or not. In all three solvers some in-processing steps needed to be removed or slightly altered, or enforce an early downgrading of all locally watched clauses.

Acknowledgment

We want to express our gratitude towards the organizers of the SAT Competition 2021 for making such an event possible. Additionally we would like to thank Florian Schintke for his support and the IT and Data Services members of the Zuse Institute Berlin for providing the infrastructure and their fast help. Also we would like to thank the authors of Maple_LCM_Dist_ChronoBT and everyone else contributing to this solver. In particular, we would like to thank the authors of [4] who discuss a different alteration of the propagation order and from whom we borrowed the idea of restricting our approach to clauses with small LBD. Benjamin Kaiser thanks Marc Hartung for introducing him to this wonderful subject of SAT-Solving and being a marvelous mentor during the past 18 months.

References

[1] Vadim Ryvchin and Alexander Nadel, “Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking” in Proceedings of SAT Competition, 2018 p.29
[2] Benjamin Kaiser and Robert Clausecker, “CleanMaple” in Proceedings of SAT Competition, 2021 if accepted for submission
[3] Armin Biere, Katalin Fazekas, Mathias Fleury, Maximilian Heisinger and Johannes Kepler, “CADICAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING Entering the SAT Competition 2020” in Proceedings of SAT Competition, 2020 p.50
[4] Jingchao Chen “Core First Unit Propagation” in arXiv, cs.LO, 2019, 1907.01192

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

hKis, hCaD, PaKis and PaInleSS_ExMapleLCMDistChronoBT in the SC21

Rodrigue Konan Tchinda
Department of Mathematics and Computer Science
University of Dschang
Dschang, Cameroon
rodriguekonanktr@gmail.com

Clémentin Tayou Djamegni
Department of Mathematics and Computer Science
University of Dschang
Dschang, Cameroon
dtayou@gmail.com

***Abstract*—This document describes the sequential solvers hKis, hCaD and the parallel solvers PaKis and PaInleSS_ExMapleLCMDistChronoBT submitted to the 2021 SAT Competition.**

I. Introduction

The results of the latest SAT competition showed very good performances of the sequential solvers Kissat [1] and CaDiCaL [1], [2] in the main track. The highly optimized data structures and algorithms used by these solvers make them particularly efficient. However, this does not exclude the possibility of improvements. We proposed for the 2021 edition of the SAT competition, versions of these solvers that integrate the PSIDS heuristic [3] for choosing polarities of decision variables. Moreover, we submitted a parallel version of Kissat built on top of the Painless framework [4] as well as a slightly modified version of PaInleSS_ExMapleLCMDistChronoBT [5].

II. hKis and hCaD

hKis and hCaD are both "hacks" of Kissat and CaDiCaL [1], [2] respectively. They all integrate the PSIDS heuristic [3] for selecting a polarity once a branching variable chosen. PSDIS is enabled through the Boolean option `--psids`. These solvers also change the default configuration of the base solvers as follows:

- hCaD is submitted with two configurations:
  - `default` where psids=0, target=2, chrono=0 and walk=0;
  - `psids` where psids=1, target=2, chrono=1 and walk=0.
- hKis is submitted with three configurations:
  - `default` where psids=0, target=2 and chrono=0;
  - `psids` where psids=1, target=2 and chrono=0;
  - `unsat` where psids=0, target=1, walkinitially=1 and chrono=1.

The `default` configuration of hCaD is submitted to the new CaDiCaL Hack subtrack of the 2021 SAT Competition.

III. PaInleSS_ExMapleLCMDistChronoBT

The parallel solver PaInleSS_ExMapleLCMDistChronoBT is identical to that we submitted to the 2020 SAT Competition [5] except for a slight change where we now load the input formula to the workers in parallel. The version we submitted to the 2021 SAT Competition was configured to launch 24 workers in parallel.

IV. PaKis

In sequential SAT solvers, it is unlikely to find a single configuration that is the most efficient on all benchmarks of a given set. There are instances that can be easily solved by specific configurations of a given solver that are not necessarily its best configuration. Hence, running multiple configurations of a solver in parallel may help speedup solving times. The goal of PaKis is to select a number of configurations in order to approximate the performance of the "Best Virtual Configuration" of the solver Kissat. To achieve this, we used a test set consisting of the new instances submitted to the SAT Race 2019 and selected among a large number of possible configurations those that had the best results for SAT, UNSAT and SAT+UNSAT. Table I gives the configurations for the 24 workers used in PaKis. The meaning of each of the options in this table can be obtained by running the solver Kissat with the traditional `--help` option.

In contrast to many parallel SAT solvers, PaKis does not allow any information sharing. This has some advantages regarding for instance the determinism of the solver and the production of DRAT proofs.

V. Acknowledgments

We would like to thank the developers of MapleLCMDistChronoBT [6], MapleLCMDistChronoBT-DL [7], PaInleSS [4], Kissat and CaDiCaL [1].

TABLE I
PAKIS WORKERS' CONFIGURATIONS

Id	tier1	chrono	stable	walkinitially	target	phase
0	2	1	1	0	1	1
1	2	1	1	0	2	1
2	2	1	0	0	1	1
3	2	0	1	0	1	1
4	2	0	1	0	1	0
5	2	1	1	0	1	0
6	2	0	2	0	1	1
7	2	1	1	0	0	1
8	2	0	1	0	0	0
9	2	1	1	0	0	0
10	2	1	1	1	1	1
11	2	0	1	0	2	1
12	2	0	1	0	2	0
13	3	0	1	0	2	0
14	3	0	1	0	2	1
15	2	1	1	0	2	0
16	2	0	2	0	2	1
17	2	1	1	0	0	1
18	2	0	1	0	0	0
19	2	1	1	0	0	0
20	3	1	1	0	0	1
21	3	1	1	0	2	1
22	2	1	1	1	2	1
23	2	0	0	0	1	1

REFERENCES

[1] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[2] A. Biere, "CaDiCaL at the SAT Race 2019," in *Proc. of SAT Race 20219 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.

[3] R. K. Tchinda and C. T. Djamegni, "Padc maplelcmdistchronobt, padc maple lcm dist and psids maplelcmdistchronobt in the sr19," *SAT RACE 2019*, p. 33.

[4] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Painless: a framework for parallel sat solving," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2017, pp. 233–250.

[5] R. Konan Tchinda and C. Tayou Djamegni, "Exmaplelcmdistchronobt, upglucose-3.0 padc and painless exmaplelcmdistchronobt in the sc20," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 17–18.

[6] V. Ryvchin and A. Nadel, "Maple_lcm_dist_chronobt: Featuring chronological backtracking," *SAT COMPETITION 2018*, p. 29, 2018.

[7] S. Kochemazov, O. Zaikin, V. Kondratiev, and A. Semenov, "Maplelcmdistchronobt-dl, duplicate learnts heuristic-aided solvers at the sat race 2019," *SAT RACE 2019*, p. 24.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

CaDiCaL Modification – Watch Sat

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

***Abstract*—The solver CaDiCaL is different from other participants in SAT competitions in many aspects. Porting an algorithm detail from CaDiCaL to MergeSAT resulted in a performance degradation. Hence, this solver modification brings CaDiCaL's behavior closer to other solvers again: when watching a satisfied literal during unit propagation, the clause is moved to the watch list of that literal. Previously, CaDiCaL just updated the blocking literal of the clause and kept the clause in the current watch list. The solver CaDiCaL-watch-sat watches the satisfied literal.**

I. Unit Propagation Improvements

SAT solvers are used in many fields. Hence, some solvers are heavily tuned to perform well for the target application. Other research focusses on improving the overall solver performance in general. Many heuristic and algorithmic extensions to the core algorithm have been proposed [1]. The overall runtime distributions among the algorithm components still did not change significantly: unit propagation still takes a vast majority of the overall runtime [6], [3].

A. Watching Clauses in Propagation

This modification alters an implementation detail of unit propagation that is different in CaDiCaL when being compared to other SAT solvers that participate in competitive events. The two watched literals scheme has been implemented first in [7]. The next major improvement to skip processing clauses early was to move literals from the clause into the watch list data structure, so called *blocking literals*. MiniSat 2.2 2.1 [2] started to use a blocking literal. When propagating a clause, first the current truth value blocking literal is checked. In case the blocking literal is satisfied, the related clause is known to be satisfied. Therefore, the clause does not have to be processed further. This technique helps to improve the performance of the SAT solver [6].

In MiniSat 2.2, the blocking literal of a clause is typically the other watched literal. However, any other literal of the clause could be chosen.

B. How to Handle Satisfied Clauses

When a blocking literal is not satisfied, the clause has to be processed. During this process, each clause of the watch list for the current literal has to be iterated. For each clause, the truth value of all literals has to be checked, in case we find a *conflict clause* or *unit clauses* that force the extension of the current truth assignment. For satisfied clauses, we only need to process the literals until we find a satisfied clauses.

One difference between CaDiCaL and MiniSat 2.2 based solvers is the way how they treat these satisfied clauses. MiniSat 2.2 based solvers watch the satisfied literal. CaDiCaL implements further extensions, like memorizing the literal in a clause that was tested when last processing the clause [4].

a) Always Watching the Satisfied Literal: When a satisfied literal is detected in a clause during propagating a literal, the clause is removed from the current watch list. As a next step, solvers append the clauses to the watch list of the satisfied literal. Both operations are constant time, but require accessing the other watch list, which can lead to a cache miss [6] and TLB miss [3]. The watch list of the other literal can be higher in the search tree, so that the clause will be touched less frequent in the remainder of the search. Restart might reduce the saving, on the other hand solver today use *partial restarts* [9], *chronological backtracking* [8] as well as *trail saving* [5]. All these technique give this saving back partially.

This approach is implemented by MiniSat 2.2 based solvers.

b) Just Update the Blocking Literal: As an alternative, CaDiCaL keep watching the current literal, which is now falsified, but updates the blocking literal to the satisfied literal. While this breaks the assumption that falsified literals are only watched for *conflict clauses* or *unit clauses*, we still know that the clause is satisfied. Hence, breaking this assumption does not have consequences. The positive effect is that the clause does not have to be removed from the current watch list. This results in no cache miss, nor a TLB miss. However, when the search progresses, after backtracking, the same clause might need to be processed again. In case the satisfied literal is still satisfied, only the blocking literal has to be processed. Otherwise, backtracking also removed the assignment for the blocking literal, so that the whole clause needs to be processed again.

c) Watching the Satisfied Literal in CaDiCaL*:* Preliminary testing with MergeSAT when just updating the blocking literal of a clause resulted in a performance degradation. Hence, removing this technique for CaDiCaL might result in a performance improvement. The solver CaDiCaL-watch-sat implements this modification.

Not processing a satisfied clause during propagation soon again can result in a different order of propagated literals, as well as different conflicts, and consequently in different heuristic updates and many different follow-up search steps of the solver. Hence, performance differences can not only be attributed to less or more compute resource utilization.

II. Availability

The source of the modified CaDiCaL is publicly available at https://github.com/conp-solutions/cadical/tree/watch-sat. The used version of the tool is "rel-1.4.0-1-gc09aa31".

References

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. Amsterdam: IOS Press, 2009.

[2] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Heidelberg: Springer, 2004, pp. 502–518.

[3] J. K. Fichte, N. Manthey, J. Stecklina, and A. Schidler, "Towards faster reasoners by using transparent huge pages," in *Principles and Practice of Constraint Programming*, H. Simonis, Ed. Cham: Springer International Publishing, 2020, pp. 304–322.

[4] I. P. Gent, "Optimal implementation of watched literals and more general techniques," *J. Artif. Intell. Res.*, vol. 48, pp. 231–251, 2013. [Online]. Available: https://doi.org/10.1613/jair.4016

[5] R. Hickey and F. Bacchus, "Trail saving on backtrack," in *Theory and Applications of Satisfiability Testing – SAT 2020*, L. Pulina and M. Seidl, Eds. Cham: Springer International Publishing, 2020, pp. 46–61.

[6] S. Hölldobler, N. Manthey, and A. Saptawijaya, "Improving resource-unaware SAT solvers," ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Heidelberg: Springer, 2010, pp. 519–534.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC 2001*. New York: ACM, 2001, pp. 530–535.

[8] A. Nadel and V. Ryvchin, "Chronological backtracking," in *Theory and Applications of Satisfiability Testing – SAT 2018*, O. Beyersdorff and C. M. Wintersteiger, Eds. Cham: Springer International Publishing, 2018, pp. 111–121.

[9] P. van der Tak, A. Ramos, and M. Heule, "Reusing the assignment trail in cdcl solvers," *JSAT*, vol. 7, no. 4, pp. 133–138, 2011.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

MergeSAT 3.0

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

***Abstract*—MERGESAT mission is to simplify keeping up with SAT development from a SAT user perspective. MERGESAT supports incremental solving, as well as being used as a library with various interfaces. MERGESAT is setup to simplify merging solver contributions into one solver. This setup should allow collaboration among solver developers more easily. The sequential SAT solver MERGESAT is a fork of the 2018 competition winner. The tool adds known as well as novel implementation and search improvements, typically based on solver contributions of previous annual competitions.**

I. THE SEQUENTIAL SOLVER MERGESAT

Most SAT solvers change over time, or their maintenance is dropped. From a user perspective, once choosing a SAT backend mostly results in fixing the performance of the selected SAT backend. Migrating between solvers is challenging. Furthermore, more recent successful solvers in SAT competitions do not support being used as a library, not incremental SAT solving. MERGESAT bridges this gap: the solver implements the broadly used MINISAT 2.2 interface, supports incremental solving, but also comes with the most recent solver improvements that have been developed in 2020.

MERGESAT is based on the SAT competition winner of 2018, MAPLE_LCM_DIST_CHRONOBT [7]. Based on recent SAT research, several known techniques as well as novel ideas has been integrated. To make continuing the long list of work that influenced MERGESAT simpler, MERGESAT uses git to combine changes from different solvers. Furthermore, MERGESAT comes with continuous integration to simplify extending the solver further. Starting in 2020, code style was enforced during CI as well, allowing to understand modifications better. MERGESAT has been tested as incremental SAT backend of several tools already. Furthermore, MERGESAT implements the SAT interface for the parallel HORDESAT solver [1], and can hence be used as a portfolio-parallel SAT solver with clause sharing as backend in HORDESAT.

A. Extensions since 2020

MERGESAT participated in SAT competition 2020 [5]. This section only details the differences since this version. All extensions can be disabled via the command line interface. Furthermore, the parameters to the solver can be specified via an environment variable. This setup allows to tune the solver even if it is used as a library in the backend.

The following improvements have been integrated into MERGESAT since the 2020 submissions:

- parsing multiple compression formats
- trail saving [3]
- use CCNR to participate in solving [8], but initialize the SLS engine lazily
- use rephasing as done in [8]
- allow to remove more clauses, as proposed in [4], but with a back-off strategy
- support printing the PCS format for configuration

Furthermore, formula simplification, as well as the CCNR engine are disabled in case too large formulas are used. When combining several extensions to the search also uncovered corner cases in the original implementation taken from the original tools. These corner cases have been addressed in MERGESAT to result in a safe search again. A full description of the solver is [6].

B. Continuous Testing

The submitted version of MERGESAT compiles on Linux and Mac OS. GitHub allows to use continuous testing, which essentially build MERGESAT, and tests basic functionality: i) producing unsatisfiability proofs, ii) building the starexec package and producing proofs and iii) solving via the IPASIR interface. All these steps are executed by executing the script "tools/ci.sh" from the repository, and the script can be used as a template to derive similar functionality.

C. Availability

The source of the solver is publicly available under the MIT license at https://github.com/conp-solutions/mergesat. The submitted starexec package can be reproduced by running "./scripts/make-starexec.sh" on this commit.

D. Submitted MERGESAT Configurations

The git version "v3.0-13-g76cb34f" has been submitted to the competition. The solver has been submitted to all sequential tracks, including the incremental track. For the main and agile tracks, a configuration *unsat* has been submitted as well. This configuration disabled the CCNR engine as well as rephasing, as experiments showed that these two modification – at least in their current form – degrade the performance on unsatisfiable formulas. Similarly, the configuration *nosimp* disables formula simplification. Comparing the results of the default configuration and *nosimp* should allow to motivate implementing formula simplfication lazily, i.e. eventually starting search without simplification as done in other solvers already [2].

II. The Parallel Solver Merge-HordeSat

HordeSat supports multi-threaded, as well as distributed solving with various SAT backends. Learned clauses can be shared among the solver instances. Merge-HordeSat extends HordeSat with integrating MergeSAT. The following modifications have been applied on top:

- share learned clauses lazily, and not interrupting SAT engines, as that misleads the original search below sequential performance
- automatically choosing to run on half the available CPU cores, to allow simple parallel use
- support to print the solver model in the multi-threaded case, to get a full parallel solver
- use MergeSAT's CNF parser that supports compressed CNF files, for more flexible use
- support to build the solver with a single "make" invocation, to simplify usage
- add MergeSAT via *git submodules*, to allow stable version tracking when benchmarking and distributing
- remove all other SAT backends, as MergeSAT based additions have not been provided

Interestingly, exchanging the parser of HordeSat resulted in a much faster file reading process. To make the solver prepared for future changes, and simple continuous testing setup has been created as well.

A. Availability

The source of the solver is publicly available under the MIT license at https://github.com/conp-solutions/hordesat.

B. *Submitted* Merge-HordeSat *Configurations*

The wrapper package to allow to use Merge-HordeSat in AWS (as done in the competition), can be found at: https://github.com/conp-solutions/hordesat-aws. The same version of the tool has been submitted to both the parallel as well as the cloud track. The solvers HordeSat, as well as its backend MergeSAT, are integrated into the code base via *git submodules*, to allow stable version tracking. The version of Merge-HordeSat is "v1.0". MergeSAT is used in version "v3.0-13-g76cb34f", as submitted to the sequential tracks.

Acknowledgment

The author would like to thank the developers of all predecessors of MergeSAT, and all the authors who contributed the modifications that have been integrated. Furthermore, without the work of the authors of HordeSat, creating Merge-HordeSat would not have happened.

References

[1] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," in *Theory and Applications of Satisfiability Testing – SAT 2015*, M. Heule and S. Weaver, Eds. Cham: Springer International Publishing, 2015, pp. 156–172.

[2] A. Biere, "PrecoSAT system description," http://fmv.jku.at/precosat/preicosat-sc09.pdf, 2009.

[3] R. Hickey and F. Bacchus, "Trail saving on backtrack," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, ser. Lecture Notes in Computer Science, L. Pulina and M. Seidl, Eds., vol. 12178. Springer, 2020, pp. 46–61.

[4] S. Kochemazov, "Improving implementation of SAT competitions 2017-2019 winners," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, ser. Lecture Notes in Computer Science, L. Pulina and M. Seidl, Eds., vol. 12178. Springer, 2020, pp. 139–148.

[5] N. Manthey, "MergeSAT," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, p. 40.

[6] ——, "The MergeSAT Solver," in *Theory and Applications of Satisfiability Testing - SAT 2021*, 2021, submitted.

[7] V. Ryvchin and A. Nadel, "Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking," in *Proceedings of SAT Competition 2018*, 2018. [Online]. Available: http://hdl.handle.net/10138/237063

[8] X. Zhang and S. Cai, "Relaxed backtracking with rephasing," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 15–15.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

PARAFROST at the SAT Race 2021*

Muhammad Osama and Anton Wijs
Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

I. INTRODUCTION

This paper presents a brief description of our solver PARAFROST which stands for *Parallel Formal Reasoning On SaTisfiability* in 2 different configurations. Compared to the solver submission in the last year competition [1], it is completely redesigned from scratch based on our recent work in [2]. It is a parallel SAT solver with GPU-accelerated inprocessing capable of harnessing NVIDIA CUDA-enabled GPUs in applying modern simplification techniques in parallel. The CDCL search is built from scratch with various optimisations based on CADICAL [3] heuristics. The inprocessing engine extends our previous work in [4], [5] with space-efficient data structures, parallel garbage collection and more. However, all submitted versions of the solver are single-threaded.

PARAFROST provides easy-to-use infrastructure for SAT solving and/or inprocessing with optimized data structures for both CPU and GPU architectures. The solver can run under Linux and Windows operating systems. The *PARAllel* keyword in PARAFROST intuitively means that SAT simplifications can be fully executed on variables in parallel as described in [2] using the Least Constrained Variable Elections (LCVE) scheduler [4], [5]. Moreover, via the Multiple Decision Making (MDM) procedure [6], the solver is capable of making multiple decisions that can be assigned and propagated at once. In principle, choosing variables to simplify or decide relies heavily on *freezing* variables, hence the name PARAFROST. The scheduled variables are *mutually independent* according to some logical properties.

II. DECISION MAKING

Based on our work in [2], the solver improves the decision heuristics of the solver submitted last year by adopting another decision queue called Variable Move To Front (VMTF) [7] where the score of a variable is defined as the number of conflicts in which the variable was involved. VMTF is implemented in CADICAL and our solver with a doubly-linked list. At the decision making step, we apply both VSIDS and VMTF for the selection of multiple decisions [8]. One can alternate between VSIDS and VMTF queues based on the restart *mode* [3] in a ping-pong manner. In CADICAL, different restart sequences are interleaved together to remedy the shortcomings of each individual one and alleviate the strengths of all. The idea is to start with the *geometric* [9] style with less frequent restarts (i.e. called in CADICAL the *stable* mode) then switch to a more aggressive style using dynamic restarts [10], [11] after some interval based on the number of conflicts. The interval is increased using the total number of propagations. In the KISSAT solver [3], the propagation metric was replaced by estimating the number of cache lines accessed by the watches in the unit propagation procedure. Currently, in PARAFROST, we adopt the same technique assuming a more realistic line size of 64-bit. Moreover, we observed that clauses are extensively checked for being *deleted* or not during propagation, simplifications, and garbage collection. To avoid dereferencing a clause only to check its state, a stencil array is created explicitly as part of the CNF data structure inspired by our parallel garbage collector proposed for the GPU solver [2].

Making a decision can always lead to conflicts, but making more of them increases the likelihood of a conflict occurring. To avoid repeatedly selecting sets of decisions that cause conflicts, they are constructed in such a way that it is guaranteed that no conflicts will occur. However, multiple decisions cannot be always selected, as the production of implications cannot indefinitely be avoided. The main question is therefore when MDM should be applied. Per search, MDM is called a number of decaying *rounds* (default is 3) if there are enough free variables to assign. If no rounds are left, it can be reset again to the initial value (e.g. 3), periodically based on conflicts, in an $n\ log(n)$ increasing step [8].

To further strengthen MDM, we add an implementation for local search using the WALKSAT strategy [12]. Besides running the local search frequently to improve the decision phases [3], we call it in MDM regularly per first round at the top level to improve the quality of the multiple decisions picked [8]. Our WALKSAT version is powered by a random number generator based on the *Xorshift32* technique discovered by George Marsaglia. The initial decision phases are assumed to be negative which goes back to MINISAT. Regarding clause minimization, we still keep the strengthening method with binaries from the previous submission which is crucial when MDM is turned on.

III. INPROCESSING

Recently, we applied GPUs in SAT solving to accelerate *preprocessing* [4], [5] and *inprocessing* [2]. In these operations, a given SAT formula is simplified, i.e., it is rewritten to a formula with fewer variables and/or clauses, while preserving satisfiability, using various simplification rules. In

* This work is part of the GEARS project with project number TOP2.16.044, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

preprocessing, this is only done once before the solving starts, while in inprocessing, this is done periodically during the solving. PARAFROST supports bounded variable elimination (BVE) [13], backward subsumption elimination (SUB) [14], Blocked Clause elimination (BCE) [15], and a new simplification technique called *eager redundancy elimination* (ERE) [2]. BVE eliminates variables by either applying the *resolution rule* or *substitution* (also known as gate equivalence reasoning) [14], [16]. Substitution detects patterns encoding logical gates, and substitutes the involved variables with their gate-equivalent counterparts. Previously [1], we only considered AND gates. In the current inprocessor, we add support for *Inverter*, *If-Then-Else* and *XOR* gate extractions. For all logical gates, substitution can be performed by resolving non-gate clauses (i.e., clauses not contributing to the gate itself) with gate clauses [16]. However, for the inverter gates we do substitution in-place without adding new clauses to the formula which saves time and memory.

ERE is a new elimination technique that we propose, which repeats the following until a fixpoint has been reached: for a given formula $\mathcal{S}$ and clauses $C_1 \in \mathcal{S}, C_2 \in \mathcal{S}$ with $x \in C_1$ and $\bar{x} \in C_2$ for some variable x, if there exists a clause $C \in \mathcal{S}$ for which $C \equiv C_1 \otimes_x C_2$, then let $\mathcal{S} := \mathcal{S} \setminus \{C\}$. In this work, we restrict removing C to the condition (C_1 is $learnt \vee C_2$ is $learnt$) $\implies$ C is $learnt$. If the condition holds, C is called a *redundancy* and can be removed without altering the original satisfiability.

In this submission, a sequential implementation of all simplifications described above is provided as part of PARAFROST. Moreover, the inprocessing engine uses a 20-byte data structure to store a clause (with at least one literal) different from the solver side which requires 24 bytes. By default, all simplifications are enabled except for BCE which was not effective in practice. The `ve+` option is always enabled with number of `phases` set to 5. The `phases=<n>` option applies `ve+` for a configured number of iterations, with increasingly large values of the threshold μ (maximum number of occurrences of a variable). If there are any unit clauses produced along the simplification process, their propagation is delayed and run in the next phase. Finally, at the last phase, the ERE method is executed once. Inprocessing is scheduled periodically based on the function $n \log^2(n)$ when at least 4,000 of the fixed (root) variables are removed. Forward subsumption is scheduled on all clauses with the same scaling function but applied within the learnt-clauses reduction procedure.

The solver version evaluated in [2] was missing important inprocessing techniques (thanks to Armin Biere for pointing this out) such as probing [17], autarky reasoning [3], and vivification [18]. In the latter, binary clauses are considered for the histogram but ignored in the actual vivification. This gives proper indication of which literals are more important to vivify. Regarding autarky, we remove autarkic variables in our implementation as in KISSAT but treat them as fixed roots (i.e. make the solver think they are deduced in the search) in both solution reconstruction and variable mapping. This saves memory and time spent in storing these variables and their satisfied clauses as witnesses. Autarky is applied only once after local search. With this submission we add them all to PARAFROST and are enabled by default.

IV. THREATS TO VALIDITY

Incorrect values of literals and variables due to ill logic or type casting breaks the solver fidelity. Therefore, PARAFROST always checks the invariants ($0 < x \leq m$) and ($1 < \ell \leq 2 \times m$) as preconditions, where m is the number of variables in the formula and ℓ encodes the variable x by a logical shift to the left. The least significant bit represents the sign. The generated model for satisfiable formulas can be verified against the original formula by enabling the `modelverify` option. The generation of DRAT proofs is also supported for the sequential solver.

V. SUBMISSIONS

The solver instance PARAFROST comprises all configurations described in the previous sections, in which MDM with local search, and all simplifications are enabled. The second configuration submitted is called PARAFROST-NOMDM which disables MDM using the command `mdmrounds=0`. The initial settings of the PARAFROST instance have been tuned on the DAS-5 cluster [19] and the Dutch national supercomputer CARTESIUS.

REFERENCES

[1] M. Osama and A. Wijs, "ParaFROST, ParaFROST CBT, ParaFROST HRE, ParaFROST ALL at the SAT Race 2020," *SAT Competition 2020*, pp. 42–43, 2020.

[2] M. Osama, A. Wijs, and A. Biere, "SAT Solving with GPU Accelerated Inprocessing," in *TACAS 2021, Luxembourg, 2021, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 12651. Springer, 2021, pp. 133–151.

[3] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *SAT Competition 2020*, 2020, pp. 51–53.

[4] M. Osama and A. Wijs, "Parallel SAT Simplification on GPU Architectures," in *TACAS*, ser. LNCS, vol. 11427. Cham: Springer International Publishing, 2019, pp. 21–40.

[5] ——, "SIGmA: GPU Accelerated Simplification of SAT Formulas," in *iFM*, ser. LNCS, vol. 11918. Springer, 2019, pp. 514–522.

[6] M. Osama and A. Wijs, "Multiple Decision Making in Conflict-Driven Clause Learning," in *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020, pp. 161–169.

[7] A. Biere and A. Fröhlich, "Evaluating CDCL Variable Scoring Schemes," in *SAT*, ser. LNCS, vol. 9340. Springer, 2015, pp. 405–422.

[8] M. Osama and A. Wijs, "Improving Decision Making in CDCL SAT Solvers," in *Journal of Automated Reasoning*, 2021, to be submitted.

[9] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *SAT*, ser. LNCS, vol. 2919. Springer, 2004, pp. 502–518.

[10] G. Audemard and L. Simon, "Refining Restarts Strategies for SAT and UNSAT," in *Principles and Practice of Constraint Programming*, M. Milano, Ed. Springer, 2012, pp. 118–126.

[11] A. Biere and A. Fröhlich, "Evaluating CDCL Restart Schemes," in *Proceedings of Pragmatics of SAT 2015 and 2018*, ser. EPiC Series in Computing, D. L. Berre and M. Järvisalo, Eds., vol. 59. EasyChair, 2019, pp. 1–17.

[12] B. Selman and H. A. Kautz, "An Empirical Study of Greedy Local Search for Satisfiability Testing," in *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, USA, 1993*. AAAI Press / The MIT Press, 1993, pp. 46–51.

[13] S. Subbarayan and D. K. Pradhan, “NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances,” in *SAT*, ser. LNCS, vol. 3542. Springer, 2004, pp. 276–291.
[14] N. Eén and A. Biere, “Effective Preprocessing in SAT Through Variable and Clause Elimination,” in *SAT*, ser. LNCS, vol. 3569. Springer, 2005, pp. 61–75.
[15] T. Balyo, A. Fröhlich, M. J. H. Heule, and A. Biere, “Everything You Always Wanted to Know about Blocked Sets (But Were Afraid to Ask),” in *SAT*, C. Sinz and U. Egly, Eds. Cham: Springer International Publishing, 2014, pp. 317–332.
[16] M. Järvisalo, M. J. Heule, and A. Biere, “Inprocessing Rules,” in *IJCAR*, ser. LNCS, vol. 7364. Springer, 2012, pp. 355–370.
[17] I. Lynce and J. Marques-Silva, “Probing-based preprocessing techniques for propositional satisfiability,” in *ICTAI*. IEEE, 2003, pp. 105–110.
[18] C. Piette, Y. Hamadi, and L. Saïs, “Vivifying Propositional Clausal Formulae,” in *ECAI*. NLD: IOS Press, 2008, pp. 525–529.
[19] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, “A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term,” *IEEE Computer*, vol. 49, no. 5, pp. 54–63, 2016.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

MapleSSV SAT Solver for SAT Competition 2021

Saeed Nejati*‡, Md Solimul Chowdhury†‡, Vijay Ganesh*
* University of Waterloo, Waterloo, Canada
† University of Alberta, Alberta, Canada
‡ Joint first authors

***Abstract*—This document describes the SAT solver MapleSSV, as part of which we implemented a set of heuristics that are found to be useful in solving SAT benchmarks that encode ARX (Addition-Rotation-XOR) functions. These heuristics are inspired by machine learning-based optimization methods, namely, improving branching using exploration, Bayesian moment matching based search initialization, and multi-armed bandit based restarts.**

I. Introduction

We present the SAT solver MapleSSV, which is a modification of the SAT solver MapleLCMDistChronoBT [1] (winner of the SAT competition 2018). There are four main modifications that we made to the base solver, enhancing branching, search initialization, restarts, and pre-processing. First, we used *exploration* in phases that the solver goes into a conflict depression (large sequences of decisions without learning any clauses) to get the solver to a more fruitful sub-space. Second, we used a Bayesian moment matching formulation of SAT to arrive at a promising initial search point, initializing variable order and polarities. Third, we employed multi-armed bandit based restarts to adaptively choose between restart strategies. Finally, we added XOR pre-processing to simplify the formula. Sections II, III and IV, describes each of these additions in more detail.

Motivation for Machine Learning-based Solver Heuristics: While a Boolean SAT solver is a decision procedure that decides whether an input formula is satisfiable, internally it can be seen as an optimization procedure whose goal is to minimize its run time while correctly deciding the satisfiability of the input formula. Every sub-routine in a SAT solver can be viewed either as a logical reasoning engine (i.e., a proof rules such as resolution in the case of conflict clause learning scheme or unit resolution in the case of BCP), or as a heuristic aimed at optimizing the sequencing, selection, and initialization of proof rules (e.g., variable selection, polarity selection, restarts, etc.). These optimization heuristic can in turn be implemented effectively using machine learning algorithms, since solvers are a data-rich environment. This philosophy was first articulated in the SAT 2016 paper by Liang et al. [2] on the LRB branching heuristic, has since been widely adopted and underpins many solver heuristics for branching, restarts, and initialization developed in recent years.

II. Exploration amid Conflict Depression Phases

Here we describe our exploration-based branching heuristic `expVSIDS`. This approach is based on our observation that CDCL SAT solving entails clear non-random patterns of bursts of conflicts followed by longer phases of *conflict depression (CD)* [3]. During a CD phase a CDCL SAT solver is unable to generate conflicts for a consecutive number of decisions. To correct the course of such a search, we propose to use exploration to combat conflict depression. We therefore design a new SAT solver, called *expSAT*, which uses random walks in the context of CDCL SAT solving. In a conflict depression phase, random walks help identify more promising variables for branching. As a contrast, while exploration explores *future* search states, LRB and VSIDS relies on conflicts generated from the *past* search states. In [3], we proposed `expVSIDS`, the exploration based extension of VSIDS. In addition to `expVSIDS`, our submitted solver MapleSSV, uses `expLRB`, the exploration based extension of LRB.

III. Initialization Problem

Many modern branching heuristics in CDCL SAT solvers assume that all variables have the same *initial* activity score (typically 0) at the beginning of the run of a solver. However, it is well known that a solver's runtime can be greatly improved if the *initial* order and value assignment of variables is not fixed a priori but chosen via appropriate static analysis of the formula. By the term initial variable order (resp., initial value assignment), we refer to the order (resp. value assignment) at the start of the run of a solver. This problem of determining the optimal initial variable order and value assignment is often referred to as the initialization problem.

In this work, we used a solution to the initialization problem based on a Bayesian moment matching (BMM) formulation of solving SAT instances and a concomitant method we refer to as BMM-based initialization [4]. Our method is used as a pre-processing step before the solver starts its search (i.e., before it makes its first decision).

A. Bayesian Moment Matching (BMM)

The SAT problem, simply stated, is to determine whether a given Boolean formula is satisfiable. In order to reformulate the SAT problem in a Bayesian setting, we start by defining a random variable for each variable of the input formula, where $P(x = T)$ shows the probability of setting x to True in a

satisfying assignment, assuming the formula is satisfiable. We assume that each of these variables has a Beta distribution, and collectively they form our prior distribution. We have the constraint that all of the clauses must be satisfied (i.e., it is assumed that the formula is satisfiable), therefore the clauses can be seen as evidence as to how the probability distribution should look like such that they are all satisfied. We then apply Bayesian inference using each clause as evidence to arrive at a posterior distribution. Applying Bayesian inference, gives us a mixture model, and this makes the learning intractable as the number of components grows exponentially with the number of clauses. To avoid this blow up, we use the method of moments to approximate the mixture Beta distribution with a single Beta distribution.

B. BMM as a Component in CDCL SAT Solvers

We implement an approximate version of the BMM method described above to solver the initialization problem of CDCL SAT solvers, since the complete method does not scale as the size of the input formulas increase. Fortunately, this approximate method is efficient and arrives at a promising point, as it attempts to satisfy as many clauses as possible. We take this starting point and initialize the preferred polarity and activity scores of each variable of an input formula, and then let the solver complete its search. The derived posterior distribution collectively represents a probabilistic assignment to the variables that satisfies most of the clauses. For polarity initialization, we used: $Polarity[x] = False$ if $P(x = T) < 0.5$ and True otherwise. For activity initialization, we gave higher priority to variables based on the confidence that BMM has about their values, i.e., $Activity[x] = \max(P(x = T), 1 - P(x = T))$. We initialized both VSIDS and LRB scores with the aforementioned methods.

IV. Multi-Armed Bandit Restart

Many restart policies have been proposed in the SAT literature [5], [6], in particular we focus on the uniform, linear, Luby, and geometric restart policies [7]. For a given SAT instance, we can not know a priori which of the 4 restart policies will perform the best. To compensate for this, we use multi-armed bandits (MAB) [8], a special case of reinforcement learning, to switch between the 4 policies dynamically during the run of the solver. We chose to use discounted UCB algorithm [9] from MAB literature, as it accounts for the non-stationary environment of the CDCL solver, in particular changes in the learnt clause database over time. Discounted UCB has 4 actions to choose from corresponding to the uniform, linear, Luby, and geometric restart policies. Once the action is selected, the solver proceeds to perform the CDCL backtracking search until the chosen restart policy decides to restart. The algorithm computes the average LBD of the learnt clauses generated since the action was selected, and the reciprocal of the average is the reward given to the selected action. Intuitively, a restart policy which generates small LBDs receives larger rewards and UCB increases the probability of selecting that restart policy in the future. Over time, this biases UCB towards restart policies that generate small LBDs for the give input SAT instance [10].

V. Availability and License

The source code of our solver have been made freely available under the MIT license. Note that the license of the M4RI library (which is used to implement Gaussian elimination) is GPLv2+.

Acknowledgment

We thank the authors of Glucose, GlueMiniSat, Lingeling, CryptoMiniSat, and MiniSAT for making their solvers available to us and answering many of our questions over the years.

References

[1] "Maplelcmdistchronobt, http://sat2018.forsyte.tuwien.ac.at/solvers/main_and_glucose_hack."

[2] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning Rate Based Branching Heuristic for SAT Solvers," in *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT'16, 2016.

[3] M. S. Chowdhury, M. Müller, and J. You, "Guiding CDCL SAT search via random exploration amid conflict depression," in *Proceedings of AAAI 2020*, 2020, pp. 1428–1435.

[4] H. Duan, S. Nejati, G. Trimponias, P. Poupart, and V. Ganesh, "Online bayesian moment matching based sat solver heuristics," in *International Conference on Machine Learning*. PMLR, 2020, pp. 2710–2719.

[5] A. Biere, "Adaptive Restart Strategies for Conflict Driven SAT Solvers," in *Theory and Applications of Satisfiability Testing–SAT 2008*. Springer, 2008, pp. 28–33.

[6] G. Audemard and L. Simon, "Refining Restarts Strategies for SAT and UNSAT," in *Principles and Practice of Constraint Programming*. Springer, 2012, pp. 118–126.

[7] A. Biere and A. Fröhlich, "Evaluating CDCL Restart Schemes," in *Pragmatics of SAT*, 2015.

[8] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. MIT Press Cambridge, 1998, vol. 135.

[9] A. Garivier and E. Moulines, *Algorithmic Learning Theory: 22nd International Conference, ALT 2011, Espoo, Finland, October 5-7, 2011. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ch. On Upper-Confidence Bound Policies for Switching Bandit Problems, pp. 174–188.

[10] S. Nejati, J. H. Liang, C. Gebotys, K. Czarnecki, and V. Ganesh, "Adaptive restart and cegar-based solver for inverting cryptographic hash functions," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2017, pp. 120–131.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

SLIME SAT Solver

1st Oscar Riveros
PEQNP
Santiago, Chile
oscar.riveros@peqnp.science

***Abstract*—A nobel rephasing technique via Black-Box HESS algorithm, alternate BOOST Heuristic, and deterministic and simplified variation of techniques described on RelaxedLCMD-CBDLnewTech and DurianSat [1].**

I. Introduction

We improve the restart stage with a reconfiguration of preassigned polarities, via HESS black-box algorithm (Relaxed), in this case with a MaxSAT Oracle, that improve the reassignment of polarities with a approximate MaxSAT solution, based on the current assignment. We replace the pseudo-random calls with the conflicts counter, this allow a deterministic execution on each instance. We improve the BOOST Heuristic from competition 2019 and 2020, with alternate behaviour, i.e. this alternate the zone of execution according the stage of the solver VSID or not-VSID. We add a parameter "massive" used on the SLIME Cloud, that produce a initial random assignment of polarities.

II. Methods

A. HESS black-box algorithm

HESS black-box algorithm use the "The Monty Hall Problem" [2] to approximate values from an Oracle, In this case a MaxSAT oracle get a complete assignment and return the number of falsified clauses, for CDCL use the native and learnts clauses.

B. HESS º1 order (Relaxed)

- Create an initial boolean array $\rho(bit_0, bit_1 \dots\ bit_n)$ based on current assignment
- Set the current value to ∞, and i to 0.
- Change the state of i-th boolean variable
- Get $Oracle(\rho)$
  1) less than current value, reassign and retain the current assignment, and continue with next variable.
  2) if greater, change the variable to original state, and continue with next variable.
  3) if equal, reassign the polarities to ρ, and exit.
- Continue with execution of CDCL.

C. Experimental Evaluation

We select a small set of cryptography related instances (Combined SLS and CDCL instances at the SAT Competition 2020, Mate Soos) [1], and add "The State of The Art SAT Solver" Kissat-sc2020, RelaxedLCMDCBDLnewTech-sc2020 with SLIME deterministic and without HESS (presented on the competition 2021), to evaluate the performance and pure state difference.

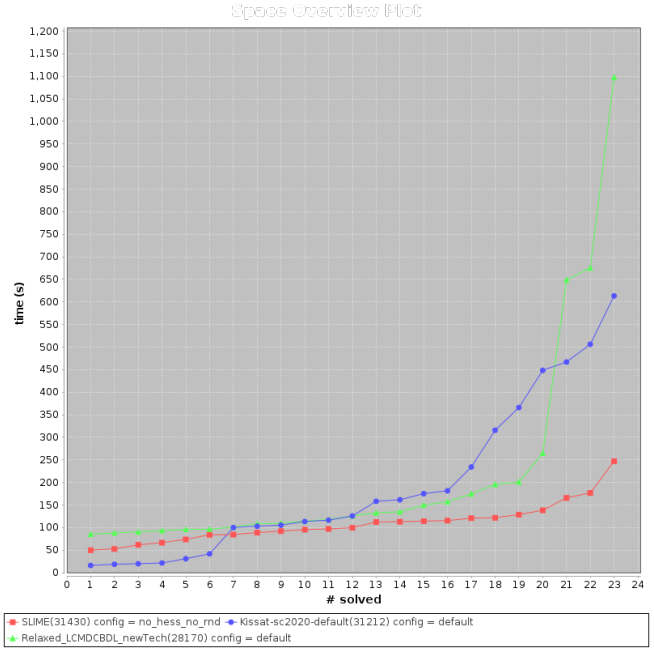


Fig. 1.

III. SLIME CLOUD

Consist on a MPI implementation of SLIME where al nodes with hess no-deterministic configuration compete for the solution, can generate certificates certificates for UNSAT.

The no-deterministic indicates that the initial polarities are random assigned.

Thanks to all supporters of http://www.peqnp.com projects.

References

[1] Balyo, T., Froleyks, N., Heule, M. J. H., Iser, M., Järvisalo, M., Suda, M. (Eds.) (2020). Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions. (Department of Computer Science Report Series B; Vol. B-2020-1). Department of Computer Science, University of Helsinki.

[2] Weisstein, Eric W. "Monty Hall Problem." From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/MontyHallProblem.html

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Mallob in the SAT Competition 2021

Dominik Schreiber
Institute of Theoretical Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
dominik.schreiber@kit.edu

***Abstract*—We describe our contribution to the parallel and cloud tracks of the SAT Competition 2021. Notable differences over last year's submission include additional diversification, a simple kind of memory awareness, lock-free clause import in Lingeling, and updated parametrization of clause sharing.**

***Index Terms*—Parallel SAT solving, distributed SAT solving**

I. Introduction

After the great success of the distributed SAT solving system named *mallob-mono* in the SAT Competition 2020 [1], we submit a new version of this system with a number of improvements to this year's SAT Competition. We submit our system not only to the cloud track but to the parallel track as well in order to see how it compares to state-of-the-art shared memory solvers at a smaller scale.

We decided to name our submission *Mallob* and omit the suffix "-mono" which was meant to emphasize the mode of operation where only one instance at a time is solved. Mallob as a whole is capable of solving many instances at once and performing decentralized malleable load balancing on top of these jobs [2]. Furthermore, since last year, Mallob gained promising new solver interfaces to CaDiCaL [3], Glucose [4], and (work in progress) MergeSAT [5]. However, due to the rules of the competition, we cannot make use of more than one CDCL solver and can only solve one instance at a time. In productive environments free of such constraints, our system can reach better performance by employing a careful mix of these solvers and by solving several instances in parallel.

II. System and Solver Setup

The setup of our system remains mostly unchanged compared to last year's submission. We subdivide each physical compute node into groups of four hardware threads each and run one MPI process on each such group. Each MPI process then runs four core solvers in parallel. We make use of Lingeling and YalSAT [3] as last year with the same kind of "native" diversification options.

We revisited the concurrent program code in Lingeling's solver interface and made the clause import lock-free by introducing a concurrent lock-free ring buffer[1] instead of a simple array guarded by a mutex. While the array used to grow indefinitely if a solver did not import any clauses for a long time, the size of the ring buffer is now limited to a small multiple of the payload that may be shared each round, and if the buffer is full, further incoming clauses are discarded.

[1]https://github.com/rmind/ringbuf

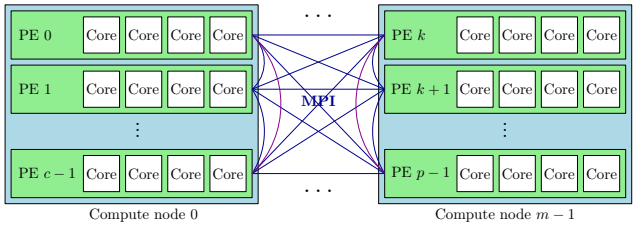


Fig. 1. Architecture of Mallob in the cloud track [2]

As an experimental change, we introduce a new kind of diversification based on permuting the input: With probability p, a solver will randomly shuffle all clauses and the literals within each clause. Preliminary experiments at a small scale showed that performing this permutation in *each* solver is detrimental to the overall performance, but we believe that letting a small ratio of solvers operate on a permutation of the input may provide different insights to the problem and improve performance in large scale environments where all native diversification options are exhausted. We set $p = 0.03$ which, in the cloud track with 1600 solver threads, leads to an expected number of 48 "permuted solvers" and still leaves the great majority of solvers running on the original problem.

III. Memory Awareness

Last year's benchmarks featured a few very large instances for which we experienced out-of-memory errors on the compute nodes running Mallob. We introduce a simple kind of memory awareness to alleviate this problem: As described in [2], we limit the total number of literals (including clause separation zeroes) which may be imported to the solver threads of a particular MPI process. This measure provides a coarse estimate on the memory the solver threads will use. If the input size were to exceed this limit, the number of solver threads to spawn is reduced such that either the import size is below the limit or only a single thread remains. We allow a total import size of $50 \cdot 10^6$ for each thread which we expect to prevent most out-of-memory issues given that in both tracks 4 GB of main memory are available per hardware thread.

IV. Clause Exchange

We repaired a subtle issue within the merging step of our distributed clause aggregation scheme: In the set data structure

used for bookkeeping inserted clauses and for detecting duplicate clauses, we did use a hash function which is insensitive to the ordering of literals [6] but checked the equality of two clauses in an order-sensitive manner. We made the equality check order-insensitive as well such that our filtering during aggregation should now reliably detect duplicates even if the literals are ordered differently.

Based on large scale experiments [2] we adjusted the clause sharing parameters of our system compared to last year's submission. We significantly increased the clause length limit from five to 30: As our clause aggregation scheme only shares the globally shortest clauses, we found that it is not particularly harmful (but can rather be beneficial) to let individual solvers export some longer clauses. Likewise, experiments indicated that at the scale of the cloud track, it is beneficial to slightly increase the overall volume of clauses which can be shared compared to last year's configuration. As a result, we increased the clause buffer discount factor α from 0.75 to 0.9. In the parallel track, we set $\alpha = 1$ because we believe that employing only 64 solvers on a single machine allows for even more clauses to be shared with less of a penalty. This is the only difference between our parallel track submission and our cloud track submission.

Acknowledgment

The author expresses his heartfelt thanks to Armin Biere for providing the core solvers of Mallob, and to Laurent Simon and Armin Biere for allowing the use of Glucose and CaDiCaL in the competition even though it was ultimately not possible to make use of them.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

Evaluation of this work was partially performed on the supercomputer ForHLR funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. The author gratefully acknowledges the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de).

References

[1] D. Schreiber, "Engineering HordeSat towards malleability: mallob-mono in the SAT 2020 cloud track," in *Proc. of SAT Competition*, pp. 45–46, 2020.

[2] D. Schreiber and P. Sanders, "Scalable SAT solving in the cloud," in *International Conference on Theory and Applications of Satisfiability Testing*, 2021. In review.

[3] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2018," *Proc. of SAT Competition*, pp. 13–14, 2018.

[4] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[5] N. Manthey, "MergeSAT," in *Proc. of SAT Competition*, pp. 40–41, 2020.

[6] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio SAT solver," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 156–172, Springer, 2015.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

New Concurrent and Distributed Painless solvers: P-MCOMSPS, P-MCOMSPS-COM, P-MCOMSPS-MPI, and P-MCOMSPS-COM-MPI

Vincent Vallade*, Ludovic Le Frioux†, Razvan Oanea*, Souheib Baarir*§, Julien Sopena*†, Fabrice Kordon*, Saeed Nejati¶, Vijay Ganesh¶
*Sorbonne Université, LIP6, CNRS, UMR 7606, Paris, France
†INRIA, Delys Team, Paris, France
‡Université Paris Nanterre, France
§University of Waterloo, Waterloo, ON, Canada

***Abstract*—This paper describes the solvers P-MCOMSPS and P-MCOMSPS-COM submitted to the parallel track of the SAT Competition 2021; as well as P-MCOMSPS-MPI and P-MCOMSPS-COM-MPI submitted to the cloud track of the SAT Competition 2021. P-MCOMSPS and P-MCOMSPS-MPI are LBD-based, and P-MCOMSPS-COM and P-MCOMSPS-COM-MPI are community and LBD-based.**

I. Introduction

P-MCOMSPS is a concurrent SAT solver built by using the Painless framework [1]. It is a portfolio-based [2] solver implementing a diversification strategy [3], fine control of learnt clause exchanges [4] based on LBD [5], using MapleCOMSPS [6] as a core sequential solver, and where learnt clause strengthening [7] has been integrated. P-MCOMSPS-COM is based on P-MCOMSPS and uses COM and LBD sharing [8]. P-MCOMSPS-MPI (resp. P-MCOMSPS-COM-MPI) is a distributed solver using on each node P-MCOMSPS (resp. P-MCOMSPS-COM), and relying on MPI for termination and clause sharing between nodes.

Section II details the implementation of P-MCOMSPS using Painless and MapleCOMSPS. Section III explains how community and LBD sharing has been implemented in P-MCOMSPS-COM. Finally, section IV explains how our concurrent solvers have been adapted to implement P-MCOMSPS-MPI and P-MCOMSPS-COM-MPI.

II. P-MCOMSPS

This section describes the overall behaviour of our competing instantiation named P-MCOMSPS. Its architecture is highlighted in Fig. 1. It implements the Painless strengthening described in [9].

A. MapleCOMSPS

MapleCOMSPS [6] has been adapted for the parallel context as follows: (1) we parametrized the solver to use either LRB [10], or VSIDS [11] (resp. L and V); (2) we added callbacks to export and import clauses; (3) we added an option to activate or not the Gaussian elimination (GE) preprocessing; (4) we parametrized the solver to use as a variable score comparator either $<$ or $<=$ (resp. head: H and tail: T).

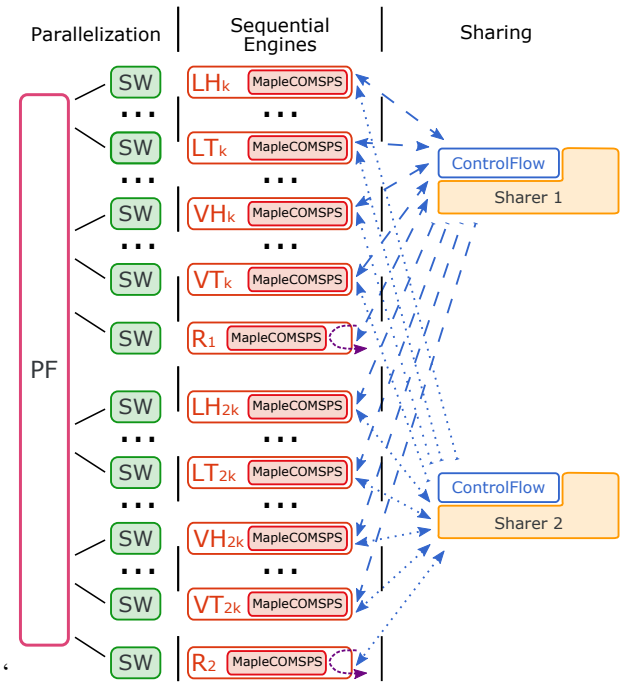


Fig. 1. Architecture of P-MCOMSPS

B. Strengthener

Two *reducer* engines (R in Fig. 1) implement the algorithm introduced in [7]. We implemented the strengthening operation as a decorator of *SolverInterface*. This decorator uses, by delegation, another *SolverInterface* to apply the strengthening, in the present case a MapleCOMSPS solver.

C. Portfolio and Diversification

As depicted in Fig. 1, P-MCOMSPS implements a portfolio strategy (PF), where two solvers are used as reducers, and the other underlying core engines are either LH, LT, VH or VT instances (i.e., combination of V or L, and H or T). For each type of instances, we apply a sparse random

diversification [3]. Moreover, only one of the solvers performs the GE preprocessing.

D. Controlling the Flow of Shared Clauses

In `P-MCOMSPS`, the sharing strategy `ControlFlow` is inspired by the one used by [3], [4]. As highlighted in Fig. 1, we instantiate two sharers, for each half of the solvers and one reducer are producers. It gets clauses from this producer and exports some of them to all others (the consumers).

The exchange strategy is defined as follows: each solver exports clauses having an LBD value under a given threshold (2 at the beginning). Every 1.5 seconds, 1500 literals (the sum of the size of the shared clauses) are selected from each producer by the sharers and dispatched to consumers. The LBD threshold of the concerned solver is increased (resp. decreased) if an insufficient (resp. a too big) number of literals are dispatched (75% and 98%).

E. Online Strengthening

There is one reducer engine that is both consumer and producer in each of the two sharing groups. It receives clauses from half of the sequential solvers, strengthened them, in case of success it then exports them back. The sharing mechanism will then share this strengthened clauses to all the other solvers. Since a strengthened clause subsumes the original one, it is likely that cores will forget the original clause over time.

III. `P-MCOMSPS-COM`

`P-MCOMSPS-COM` has exactly the same behaviour than `P-MCOMSPS` except for its clause sharing policy where clauses are filtered through community and LBD as presented in [8].

A. Community Structure

It is well admitted that real-life SAT formulas exhibit notable "structures". One way to highlight such structures is to represent the formula as a graph and analyze its community structure [12]. In `P-MCOMSPS-COM`, community structure is computed using the Louvain method [13] on the variable incident graph (VIG) of the simplified formula. The result is a disjoint partition of the variables present in the formula. Since this process can take some time, only one of the solver is responsible for the computation; until it ends sharing is based only on LBD as in `P-MCOMSPS`.

B. Community and LBD Sharing

We call *"COM of a clause"* the number of communities in which a clause spans. To compute the COM of a clause, we consider the variables corresponding to the literals of the clause and we count the number of distinct communities represented by these variables. When information on community structure is available, sharing policy switches and clauses are filtered using COM and LBD: shared clauses are those with LBD ≤ 3 or (LBD ≤ 4 and COM ≤ 3); this threshold has been highlighted in [8].

IV. `P-MCOMSPS-MPI` AND `P-MCOMSPS-COM-MPI`

This section presents `P-MCOMSPS-MPI` (reps. `P-MCOMSPS-COM-MPI`) which is a distributed adaptation of `P-MCOMSPS` (resp. `P-MCOMSPS-COM`). In order to adapt our concurrent solvers for the cloud track we added distant clause sharing, and termination. Moreover, since in the cloud track nodes have less CPUs, underlying concurrent solvers only use one reducer and one sharing group.

A. Distant Clause Sharing

On each node a concurrent solver (described in previous sections) runs. We added a component called `VirtualSolverAsynchronous` which supports the MPI-based communications between nodes. This solver receives clauses from other workers on the same node and send them to the other nodes. It also receives distant clauses (from other `VirtualSolverAsynchronous`) which are spread over the local node using the classical sharing mechanism.

B. Termination

Termination is handled by regularly synchronising main threads of each concurrent solver and is implemented thanks to the `MPI_Allgather` function.

REFERENCES

[1] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Painless: a framework for parallel sat solving," in *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 233–250, Springer, 2017.

[2] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2009.

[3] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," in *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 156–172, Springer, 2015.

[4] Y. Hamadi, S. Jabbour, and J. Sais, "Control-based clause sharing in parallel sat solving," in *Autonomous Search*, pp. 245–267, Springer, 2011.

[5] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers.," in *IJCAI*, vol. 9, pp. 399–404, 2009.

[6] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maplecomsps lrb vsids, and maplecomsps chb vsids," pp. 20–21, 2017.

[7] S. Wieringa and K. Heljanko, "Concurrent clause strengthening," in *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 116–132, Springer, 2013.

[8] V. Vallade, L. Le Frioux, S. Baarir, J. Sopena, V. Ganesh, and F. Kordon, "Community and LBD-based clause sharing policy for parallel SAT solving," in *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT'20)*, pp. 11–27, Springer, 2020.

[9] V. Vallade, L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "On the usefulness of clause strengthening in parallel sat solving," in *Proceedings of the 12th NASA Formal Methods Symposium (NFM)*, Springer, 2020.

[10] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for sat solvers," in *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 123–140, Springer, 2016.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference (DAC)*, pp. 530–535, ACM, 2001.

[12] C. Ansótegui, J. Giráldez-Cru, and J. Levy, "The community structure of sat formulas," in *int. conf. on Theory and Applications of Satisfiability Testing*, pp. 410–423, Springer, 2012.

[13] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Improving CDCL via Local Search

Xindi Zhang, Shaowei Cai*, Zhihan Chen

[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{zhangxd,caisw,chenzh}@ios.ac.cn

***Abstract*—This document describes our SAT solvers submitted to the SAT Competition 2021, the solvers are based on Relaxed, CaDiCaL and kissat.**

I. LStech_Maple

$LStech\_Maple$ (short for $LSTech$) is the improved version of $Relaxed\_LCMDCBDL\_newTech$ (short for $Relaxed$), which proposed the relaxed CDCL method [9]. The relaxed method is to relax the backtracking process for protecting promising partial assignment, where a promising assignment is defined according to its consistency and length. When the CDCL process meets some conditions, the algorithm will enter a non-backtracking stage until it gets a full assignment α. Once it gets α, a local search SAT solver is called immediately.

The differences between $LSTech$ and $Relaxed$ lie in the non-backtracking stage entrance conditions and the local search process entrance conditions. Inspired by the updating method of target phase [4], $LSTech$ enters the non-backtracking phase to construct a full assignment each time the CDCL process reaches a higher trail. And $LSTech$ enters the local search process according to the number of restarts, rather than after each non-backtracking stage. The detailed rules are described as follow.

No-backtracking Stage Entrance Rule: Let p be the size of non-conflict trail that allowing the algorithm enter the non-backtracking stage. $p = 0$ at the beginning. If the CDCL process reaches a higher no-conflict trail with size p', then it will enter the non-backtracking stage, and $p \leftarrow p'$ accordingly. Moreover, $p \leftarrow 0.9 \times p$ after each local search process.

Local Search Entrance Rule: CCAnr [3] is called every δ restarts and the best local search solution (denoted as lb_soln) is updated accordingly. δ is set to 300 initially. And $\delta \leftarrow \delta + 300$ if the lb_soln has not been improved, $\delta \leftarrow \delta - 300$ (keeping $\delta > 300$) if the lb_soln has been improved.

Furthermore, we improve the CCAnr with clause state based CC [5] instead of neighborhood based CC [3], and the latter method need to keep a neighborhood list for each variable, which is time and space consuming. In addition, $LSTech$ remove the distance branching strategy [8] in the beginning.

This work was supported by Beijing Academy of Artificial Intelligence (BAAI), and Youth Innovation Promotion Association, Chinese Academy of Sciences [No. 2017150].

* Corresponding author

II. CaDiCaL_rp

Phase saving [7] is a powerful and standard technique for modern CDCL solvers, which saves the latest polarity of each variable in a vetecr $polarity$. Rephase [4] is recent proposed to reset $polarity$ with some promising full assignments (which is also called phases). The goal of CaDiCaL_rp is to improve their base-solver $CaDiCaL$ and $kissat$ [4] by selecting the appropriate phases based on probability as $Relaxed$ [9].

III. kissat_cf

For better utilize the information of local search, we use the local search conflict frequency to enhance the VSIDS branching strategy [6]. The technique is used in $Relaxed$ [9] and $LSTech$ as well. The conflict frequency for a variable v is denoted as $freq(v)$, which is the number of steps in which it appears in at least one unsatisfied clause divided by the total number of steps of the local search process. $freq(v)$ will be updated after each local search process of kissat, in which a local search solver YalSAT [2] is embedded. Every 20 restarts, the activity of each variable v is bumped by $100 \times freq(v)$, unless there are variables eliminated in in-processing.

IV. kissat_bonus

Considering that modern branching strategies like VSIDS [6] only bump activity score for each variable based on the recency, which means that the activity scores of the variable in the conflict clause will be bumped by the same score inc, no matter which clause it is. Thus, we designed a method to take the clause quality (measured by LBD [1]) into account. We set a reward coefficient $bonus$ for each new conflict clause, and the score of each variable related to this conflict will be bumped with $inc * bonus$ instead of inc. $bonus$ is 1 when the LBD of the conflict clause is equal to the global average LBD value, and$bonus$ is 2 when the LBD is unit. And we design the $bonus$ factor as an exponential function negatively related to LBD.

References

[1] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI 2009*, pages 399–404, 2009.
[2] A. Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. *Sat competition*, 2014(2):65, 2014.
[3] S. Cai, C. Luo, and K. Su. Ccanr: A configuration checking based local search solver for non-random satisfiability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 1–8, 2015.

[4] A. B. K. F. M. Fleury and M. Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION 2020*, page 50, 2020.

[5] C. Luo, S. Cai, W. Wu, Z. Jie, and K. Su. Ccls: an efficient local search algorithm for weighted maximum satisfiability. *IEEE Transactions on Computers*, 64(7):1830–1843, 2014.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535, 2001.

[7] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299, 2007.

[8] F. Xiao, C. Li, M. Luo, F. Manyà, Z. Lü, and Y. Li. A branching heuristic for SAT solvers based on complete implication graphs. *Sci. China Inf. Sci.*, 62(7):72103:1–72103:13, 2019.

[9] X. Zhang and S. Cai. Relaxed backtracking with rephasing. *SAT COMPETITION 2020*, page 15.

BENCHMARK DESCRIPTIONS

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Benchmark Instance Selection

Markus Iser
Institute of Theoretical Informatics
Karlsruhe Institute of Technology
markus.iser@kit.edu

***Abstract*—We selected 400 benchmark instances for the Main track of SAT Competition 2021. The selected instances are composed of 300 new submitted instances and 100 instances which have already been used in previous competitions. For the Crypto track, we selected 200 instances, of which 115 are new and 85 have already been used in previous competitions.**

I. Introduction

Many researchers followed our call for benchmarks. Also our rules required each participating teams to submit at least 20 benchmark instances. Like this, we received a total of 1091 new benchmark instances. We filtered out instances which could be solved by Minisat within one minute. The remaining 952 instances were partitioned into 352 instances for the Main track and 600 instances for the Crypto track.

II. Main Track Benchmarks

The 352 new instances for the Main track came from 19 authors. We added the unused fractions of three large sets of instances by three different authors, i.e., Holten, Johnson and Schidler, which have been submitted last year. Per instance author, we randomly sampled a maximum of 13 benchmark instances. We added 18 instances which we randomly sampled from a large set of hard SMT Bitvector instances to obtain a total of 300 previously unseen benchmark instances (cf. Table I). This set constists of 104 to be satisfiable, 74 unsatisfiable, and 122 unknown benchmark instances.

We randomly sampled 35 satisfiable and 65 unsatisfiable instances used in previous competitions to obtain the final set of 400 benchmark instances. Table II display the final numbers of instances selected.

III. Crypto Track Benchmarks

We received three large sets of cryptographic instances for this and last years competition of which we could randomly sample 109 instances which have not been used before in previous competitions. We augmented this set by random selection of 91 cryptographic instances by 15 different authors which have been seen in previous competitions.

IV. Benchmark Instance Distribution

All instances are available at https://gbd.iti.kit.edu/. The GBD Benchmark Database (GBD) uses *instance identification* to make instances accessible online, to assign attributes to them, and to facilitate access to instances based on their attributes [1], [2].

Amount	Author	Family
13	Biere	Test Configuration
13	Bouvier	Petrinet Concurrency
13	Cherif	Maxsat Optimum
13	Chowdhury	Cellular Automata
13	Djamegni	Timetable
13	Heule	Hamiltonian Cycle
13	Jamali	Relational Dependencies
13	Jingchao	Prime Testing
13	Kaiser	Sliding Puzzle
13	Lagniez	KTF
13	Lester	Minimal Superpermutation
13	Manthey	At Least Two Solutions
13	Mengel	Edit Distance
13	Osama	strcmp Verification
11	Reeves	Generalized Mutilated Chessboard
2	Reeves	Generalized Pigeonhole
2	Riveros	Maximum Constraint Partition
10	Riveros	Sum of Three Cubes
13	Shunyang	Circuit Multiplier
13	Xindi	Argumentation
10	Yolcu	Mycielski Graph
13	Holten	Edge Matching
13	Johnson	Stedman Triples
13	Schidler	Hypertree Decomposition
18	Preiner	SMT Bitvector

TABLE I
Families and amounts of 300 previously unseen instances

	SAT	UNSAT	UNKNOWN	Σ
NEW	104	74	122	300
OLD	35	65	–	100
Σ	139	139	122	400

TABLE II
Amounts of old and new instances by result

References

[1] M. Iser and C. Sinz, "A Problem Meta-Data Library for Research in SAT", Proceedings of Pragmatics of SAT 2018, Oxford, UK, July 7, 2018., pp. 144–152, 2018

[2] M. Iser, C. Sinz, and L. Springer, "Collaborative Management of Benchmark Instances and their Attributes", CoRR 2020, https://arxiv.org/abs/2009.02995

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

CNF Encodings of Complete Pairwise Combinatorial Testing of our SAT Solver SATCH

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

***Abstract*—This note describes the benchmarks we have submitted to the SAT Competition 2021 encoding the existence of a list of configurations of a given size k which covers pairwise all combinations of configurations of our SAT solver SATCH.**

The submitted DIMACS files encode feasibility of a list of configurations of a given size k for complete combinatorial pairwise testing [1], [2] of one internal version of our SAT solver SATCH available at https://github.com/arminbiere/satch. The encodings were generated by the tool GENCOMBI that comes with SATCH.

The `configure` script of SATCH has many different build options. Given a test suite (in case of SATCH the included test suite) the basic idea of two-way or pairwise testing is to run the suite on all combinations of all possible pairs of configuration options under the constraint that incompatible configuration options are avoided. Thus we want to produce a smallest possible list of configuration options satisfying these criteria. The DIMACS files encode the existence of such a list for a given size k. By default we also make sure that each pair of options is not used in at least one configuration.

Beside the size k of the test set, which corresponds to the number of different configurations, the instances vary in terms of dropping certain sorting constraints for symmetry breaking (option `--unsorted` and suffix '`u`') or dropping the requirement that all pairs of features should also not occur in a least one configuration (option `--weak` and suffix '`w`').

For the considered version of SATCH there does exist a list of configurations of size $k = 20$ satisfying all criteria. For size $k = 20$ all four instances are easy to satisfy including dropping inclusion of sorting constraints ('`u`') or dropping the requirement that pairs should also not occur ('`w`'). The smaller ones are getting hard at around $k = 14$ and are expected to be all unsatisfiable.

Note that in practical use GENCOMBI generates a new CNF for each considered k and first doubles k until an instance becomes satisfiable, where solving time is limited. Then the tool decreases k trying to reduce the upper bound or to find a lower bound where solving time takes 10 times more than for the upper bound. If a new upper bound is found the process repeats. The individual instances are actually kept in memory and solving is simply resumed if necessary (the only incremental way of solving supported for SATCH at this point).

Supported by the Austrian Science Fund (FWF) under projects W1255-N23, S11408-N23 and by the LIT AI Lab funded by the State of Upper Austria.

REFERENCES

[1] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatiorial design," *IEEE Trans. Software Eng.*, vol. 23, no. 7, pp. 437–444, 1997. [Online]. Available: https://doi.org/10.1109/32.605761

[2] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 614–624. [Online]. Available: https://doi.org/10.1145/2970276.2970335

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

SAT-Competition Benchmarks Spawning from Concurrency Theory

Pierre Bouvier and Hubert Garavel
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
{pierre.bouvier,hubert.garavel}@inria.fr

***Abstract*—We present an original approach for generating Boolean formulas stemming from the decomposition of Petri nets into automata networks. Carefully chosen examples of these formulas have been proposed for the 2020 and 2021 editions of the SAT Competition.**

I. Scientific Context

Interesting Boolean formulas can be generated as a by-product of our recent work [1] on the decomposition of Petri nets into networks of automata, a problem that has been around since the early 70s. Concretely, we developed a tool chain that takes as input a Petri net (which must be ordinary, safe, and hopefully not too large) and produces as output a network of automata that execute concurrently and synchronize using shared transitions. Precisely, this network is expressed as a *Nested-Unit Petri Net* (NUPN) [2], i.e., an extension of a Petri net, in which places are grouped into sets (called *units*) that denote sequential components. A NUPN provides a proper structuring of its underlying Petri net, and enables formal verification tools to be more efficient in terms of memory and CPU time. Hence, the NUPN concept has been implemented in many tools and adopted by software competitions, such the Model Checking Contest[1] [3], [4] and the Rigorous Examination of Reactive Systems challenge[2] [5], [6], [7]. Each NUPN generated by our tool chain is *flat*, meaning that its units are not recursively nested in each other, and *unit-safe*, meaning that each unit has at most one execution token at a time.

Our tool chain works by reformulating concurrency constraints on Petri nets as logical problems, which can be later solved using third-party software, such as SAT solvers, SMT solvers, and tools for graph coloring and finding maximal cliques [1]. We applied our approach to a large collection of more than 12,000 Petri nets from multiple sources, many of which related to industrial problems, such as communication protocols, distributed systems, and hardware circuits. We thus generated a huge collection of Boolean formulas, from which we carefully selected a subset of formulas matching the requirements of the SAT Competition.

II. Structure of Formulas

Each of our formulas was produced for a particular Petri net. A formula depends on three factors:

- the set P of the places of the Petri net;
- a *concurrency relation* $\parallel$ defined over P such that $p \parallel p'$ is the two places p and p' may simultaneously have an execution token; and
- a chosen number n of units.

A formula expresses whether there exists a partition of P into n subsets P_i $(1 \leq i \leq n)$ such that, for each i, and for any two places p and p' of P_i, $p \neq p' \implies \neg\,(p \parallel p')$. A model of this formula is thus an allocation of places into n units, i.e., a valid decomposition of the Petri net. The value of n is chosen large enough so that the formula is satisfiable, i.e., at least one decomposition exists. This can also be seen as an instance of the graph coloring problem, in which n colors are to be used for the graph with vertices defined by the places of P and edges defined by the concurrency relation.

More precisely, each formula was generated as follows. For each place p and each unit u, we created a propositional variable x_{pu} that is true iff place p belongs to unit u. We then added constraints over these variables:

- For each unit u and each two places p and p' such that $p \parallel p'$ and $\#p < \#p'$, where $\#p$ is a bijection from places names to the interval $[1, \mathrm{card}\,(P)]$, we added the constraint $\neg x_{pu} \vee \neg x_{p'u}$ to express that two concurrent places cannot be in the same unit.
- For each place p, we could have added the constraint $\bigvee_u x_{pu}$ to express that p belongs to at least one unit, but this constraint was too loose and allowed $n!$ similar solutions, just by permuting unit names. We thus replaced this constraint by a stricter one that breaks the symmetry between units: for each place p, we added the refined constraint $\bigvee_{1 \leq \#u \leq min(\#p,n)} x_{pu}$, where $\#u$ is a bijection from unit names to the interval $[1, n]$.

Each formula is provided as a separate file, expressed in Conjunctive Normal Form and encoded in the DIMACS-CNF format[3].

III. Selection of Benchmarks

Using the approach presented in Sections I and II, we previously published a test suite, named VLSAT1 [8], of 100 formulas. However, VLSAT1 only contains satisfiable formulas, as it was designed for the Model Counting Competition, which seeks formulas accepting a large number of models.

[1] https://mcc.lip6.fr
[2] http://rers-challenge.org
[3] http://www.satcompetition.org/2009/format-benchmarks2009.html

For the SAT Competition, we therefore undertook the production of a different collection containing both satisfiable and unsatisfiable formulas, depending on the number of units chosen for a given Petri net. Figure 1 shows that, despite the symmetry-breaking constraints mentioned in Sect. II, satisfiable formulas are often easier to solve than unsatisfiable ones.

For the SAT 2020 Competition, we submitted 36 formulas, listed in Table I. All of them have been tagged as "interesting" by the organizers of the competition, who selected 7 satisfiable and 7 unsatisfiable formulas for the Main Track; the selected formulas are those marked with a star in this table.

TABLE I
LIST OF 2020 FORMULAS

type	*variables*	*clauses*	*type*	*variables*	*clauses*
SAT*	16,676	1,598,591	UNSAT	14,640	1,323,246
SAT	18,090	1,781,277	UNSAT*	15,440	1,409,906
SAT	20,868	2,204,462	UNSAT*	15,960	1,464,039
SAT	21,190	2,597,791	UNSAT	16,297	1,562,268
SAT	21,573	2,289,124	UNSAT	17,688	1,741,702
SAT*	24,450	2,770,239	UNSAT	20,424	2,157,568
SAT	26,606	3,191,844	UNSAT*	21,114	2,240,429
SAT	27,507	3,314,450	UNSAT	23,961	2,714,844
SAT	29,736	3,780,419	UNSAT	26,104	3,131,630
SAT*	30,744	3,925,645	UNSAT	26,988	3,251,923
SAT	33,040	4,437,242	UNSAT	29,205	3,712,921
SAT*	34,161	4,607,712	UNSAT*	30,195	3,855,554
SAT	36,518	5,166,057	UNSAT*	32,480	4,362,044
SAT*	37,758	5,364,539	UNSAT	33,582	4,529,625
SAT*	40,170	5,970,608	UNSAT*	35,929	5,082,743
SAT	41,535	6,200,014	UNSAT*	39,552	5,878,762
SAT*	57,038	10,572,502	UNSAT	40,896	6,104,639
SAT	71,816	14,478,832			
SAT	83,334	20,350,783			

For the SAT 2021 Competition, we submit 20 formulas, 10 satisfiable and 10 unsatisfiable ones, which are listed in Table II. All of them have been checked by five solvers (CaDiCal, MathSAT, MiniSAT, Kissat and Z3) in their most recent versions. We used a machine with a Xeon E5-2630 v3 and 128 GB RAM. Each satisfiable formula takes at least 35 seconds with any of these solvers. Each unsatisfiable formula takes at least 37 minutes with any of these solvers.

TABLE II
LIST OF 2021 FORMULAS

type	*variables*	*clauses*	*type*	*variables*	*clauses*
SAT	11,130	1,186,888	UNSAT	1134	26,703
SAT	11,374	1,150,943	UNSAT	1155	42,917
SAT	19,565	3,665,001	UNSAT	4424	545,056
SAT	29,736	3,780,419	UNSAT	5152	824,642
SAT	37,758	5,364,539	UNSAT	5600	1,042,700
SAT	59,204	10,973,962	UNSAT	11,280	4,223,777
SAT	67,996	13,708,722	UNSAT	11,664	5,532,624
SAT	68,760	13,862,744	UNSAT	14,280	6,781,327
SAT	69,524	14,016,766	UNSAT	14,424	7,585,190
SAT	70,288	14,170,788	UNSAT	16,788	9,021,307

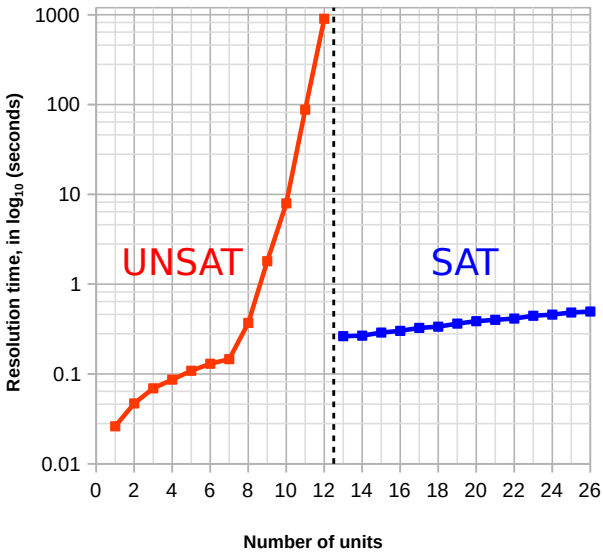


Fig. 1. Resolution times for a given NUPN

REFERENCES

[1] P. Bouvier, H. Garavel, and H. P. de León, "Automatic Decomposition of Petri Nets into Automata Networks – A Synthetic Account," in *Proceedings of the 41th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'20), Paris, France*, ser. Lecture Notes in Computer Science, R. Janicki and N. Sidorova, Eds. Springer, Jun. 2020.

[2] H. Garavel, "Nested-Unit Petri Nets," *Journal of Logical and Algebraic Methods in Programming*, vol. 104, pp. 60–85, Apr. 2019.

[3] F. Kordon, H. Garavel, L. M. Hillah, E. Paviot-Adet, L. Jezequel, C. Rodríguez, and F. Hulin-Hubard, "MCC'2015 – The Fifth Model Checking Contest," *Transactions on Petri Nets and Other Models of Concurrency*, vol. XI, pp. 262–273, 2016.

[4] F. Kordon, H. Garavel, L. Hillah, E. Paviot-Adet, L. Jezequel, F. Hulin-Hubard, E. Amparore, M. Beccuti, B. Berthomieu, H. Evrard, P. G. Jensen, D. Le Botlan, T. Liebke, J. Meijer, J. Srba, Y. Thierry-Mieg, J. van de Pol, and K. Wolf, "MCC'2017 – The Seventh Model Checking Contest," *Transactions on Petri Nets and Other Models of Concurrency*, vol. XIII, pp. 181–209, 2018.

[5] M. Jasper, M. Fecke, B. Steffen, M. Schordan, J. Meijer, J. van de Pol, F. Howar, and S. F. Siegel, "The RERS 2017 Challenge and Workshop," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN'17), Santa Barbara, CA, USA*, H. Erdogmus and K. Havelund, Eds. ACM, Jul. 2017, pp. 11–20.

[6] B. Steffen, M. Jasper, J. Meijer, and J. van de Pol, "Property-Preserving Generation of Tailored Benchmark Petri Nets," in *Proceedings of the 17th International Conference on Application of Concurrency to System Design (ACSD'17), Zaragoza, Spain*. IEEE Computer Society, Jun. 2017, pp. 1–8.

[7] M. Jasper, M. Mues, A. Murtovi, M. Schlüter, F. Howar, B. Steffen, M. Schordan, D. Hendriks, R. R. H. Schiffelers, H. Kuppens, and F. W. Vaandrager, "RERS 2019: Combining Synthesis with Real-World Models," in *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19), Part III: TOOLympics, Prague, Czech Republic*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Springer, Apr. 2019, pp. 101–115.

[8] P. Bouvier and H. Garavel, "The VLSAT-1 Benchmark Suite," INRIA Grenoble Rhône-Alpes, Tech. Rep., Nov. 2020.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Verifying Optimums of (Partial) Max-SAT Formulas

Mohamed Sami Cherif, Djamal Habet and Cyril Terrioux
Aix-Marseille Université, Université de Toulon, CNRS, LIS, Marseille, France
{mohamed-sami.cherif, djamal.habet, cyril.terrioux}@univ-amu.fr

***Abstract*—Checking whether a certain bound holds over a set of relaxation variables is a subproblem which often arises in the context of Maximum Satisfiability (Max-SAT) solving and particularly SAT-based solving. This document describes a collection of SAT instances that have been submitted to the 2021 SAT competition. These instances are derived from Max-SAT formulas whose soft clauses are augmented with relaxation variables. An At-Most-K constraint is then set over these variables to check the validity of a provided bound. We use this process to verify known solutions of (Partial) Max-SAT formulas.**

***Index Terms*—SAT, Max-SAT, At-Most-K constraint**

I. Introduction

The maximum satisfiability (Max-SAT) problem is an optimization extension of the satisfiability (SAT) problem. For a given formula in Conjunctive Normal Form (CNF), it consists in finding an assignment of the variables which maximizes the number of satisfied clauses. In Partial Max-SAT, clauses are divided into hard and soft clauses and the goal is to find an assignment that satisfies all hard clauses and maximizes the number of satisfied soft clauses. In recent years, Max-SAT solvers have achieved great breakthroughs by relying on SAT technology. In fact, Complete methods for this problem include SAT based approaches which iteratively call SAT solvers making them particularly efficient on industrial instances [5].

Checking whether a certain bound holds over a set of relaxation variables is a subproblem which often arises in the context of Maximum Max-SAT solving and particularly in SAT-based solving. For instance, Linear Search algorithms [2], [3] augment soft clauses with relaxation variables and add a CNF encoding over their sum to specify that the number of falsified soft clauses must be less than a given bound. A SAT solver is then iteratively called and the bound is increased (resp. decreased) until the formula becomes satisfiable (resp. unsatisfiable). Similarly to these algorithms, we rely on the fact that the optimum of a Max-SAT formula is the threshold in which the formula becomes satisfiable to verify the validity of a given optimum. To this end, given a Max-SAT formula and an integer value, we encode two SAT instances to check whether the given value is the threshold, i.e. the optimum of the formula.

II. Verifying (Partial) Max-SAT Optimums

Let $\phi = H \cup S$ be a Partial Max-Sat formula where H denotes the set of hard clauses and $S = \{c_1, ..., c_m\}$ the set of soft clauses. Let k be an integer value. We define the following formula:

$$\phi_k = H \cup \{c_i \cup \{r_i\} | c_i \in S\} \cup CNF(\sum_{1 \leq i \leq m} r_i \leq k)$$

where $r_1, .., r_m$ are new relaxation variables.

To verify that a given value o is the optimum of a CNF formula ϕ, it is sufficient to check that this value is the threshold in which the formula becomes satisfiable. To this end, we need to encode the formulas ϕ_{o-1} and ϕ_o and verify that ϕ_{o-1} is unsatisfiable and ϕ_o is satisfiable.

III. The Submitted Benchmark

We consider the Single machine scheduling family in the 2020 Max-SAT Evaluation described in [4]. We picked the 18 instances which were solved (by at least one solver) in the 2020 Max-SAT Evaluation and thus for which the optimum is known. For each instance ϕ, we encoded the formulas ϕ_{o-1} and ϕ_o. The submitted benchmark thus comprises 36 instances in total with 18 satisfiable instances and 18 unsatisfiable ones. We maintain the same naming conventions used in [4] except that we add '$\_sat$' or '$\_unsat$' to each formula indicating respectiviely whether it is satisfiable or unsatisfiable. We used the PySAT library [1] to add the cardinality constraints (i.e. At-Most-K constraints) over the relaxation variables. The encoding chosen for these constraints is the sequential counter encoding [6]. Finally, it is important to note that, once the constraints added, the clauses in the resulting formulas are shuffled.

References

[1] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[2] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 8, 01 2012.

[3] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability Boolean Modeling and Computation*, 7:59–64, 07 2010.

[4] Xiaojuan Liao and Miyuki Koshimura. Description of Benchmarks on Single-Machine Scheduling. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Rubens Martins, editors, *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, Department of Computer Science Report Series B 2020-2, page 54. University of Helsinki, Department of Computer Science, 2020.

[5] Antonio Morgado, Federico Heras, Mark Liffiton, Jordi Planes, and Joao Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18, 10 2013.

[6] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Safe Population Growth with Rule 30

Md Solimul Chowdhury, Martin Müller, Jia-Huai You
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada.
{mdsolimu, mmueller, jyou}@ualberta.ca

***Abstract*—A *population* is an one-dimensional grid of $n \geq 1$ organisms, where each organism evolves between being alive (1) and dead (0) across chronological time steps by following a fixed rule of evolution. At any time step $t \geq 1$, the combined states of n organisms represent the state of the population at t. At t, a population is under the *threat of extinction*, if the number of alive organisms falls below $n * (P/100)$, where $0 < P \leq 100$ and *safe*, otherwise. We say that a population *grows* over T time steps, if for any time step $t < T - 1$, population at t' has more alive organisms than population at t, with $t' = t + 1$.**

In our proposed SAT benchmark *Safe Population Growth (SPG)*, given a population of n organisms, a maximum time step T, we verify if a population could *safely grow* upto time step T, while following a fixed rule of evolution. For the SAT competition 2021, we have submitted 20 instances of the SPG benchmark.

I. SPG as a Cellular Automaton

State evolution in the Safe Population Growth (SPG) problem represents the state evolution of cells in finite elementary cellular automaton (CA) [2], with respect to (i) the *safety* constraint at any given time step and (ii) the *growth* constraint between any two consecutive time steps.

In an elementary CA, at time step $t + 1$, the state of a cell c, which has cell l (resp. r) as its left (resp. right) neighbour, is computed based on a boolean combination the states of c, l,and r at time t. There are $2^3 = 8$ combinations of boolean values for l, c, and r at t, for each of which, there are 2 ways to set the value of the state of c at $t + 1$. Hence, there are $2^8 = 256$ ways to set the new state of the c at $t + 1$. Each of these 256 ways are called *rules* [2] for a given elementary CA.

We consider *Rule 30* [3] for the SPG problem, which is known to produce chaotic patterns over time. At time $t + 1$, for a given center cell (*center*), its left (*left*) and right (*right*) neighbours, Rule 30 computes the state $center^{t+1}$ of the *center* cell as follows:

$$center^{t+1} \leftarrow left^t \ \mathbf{XOR}\ (center^t \ \mathbf{OR}\ right^t)$$

.

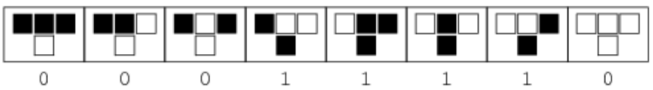


Fig. 1: State evolution for the center cell for Rule 30; black cells represents alive (1) cells, white cells represents dead (0) state.

Figure 1 (taken from [3]) shows the evolution scheme for Rule 30, which is known to exhibit chaotic behavior for some initial states. Figure 2 shows such a chaotic evolution of a CA that follows Rule 30 (also taken from [3]).

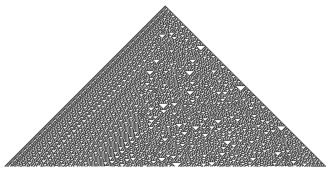


Fig. 2: Emergence of chaotic behavior with Rule 30

II. SAT encoding of the SPG problem

A. SPG as a SAT Benchmark

Given a population of $n \geq 1$ organisms, a maximum time step $T \geq 2$, , and a safety threshold $0 < P \leq 100$, the task of the SPG problem is to determine if the population evolve upto T by following Rule 30, with respect to the following two constraints:

safety: *Total number of alive organisms in every time step $1 \leq t \leq T$ is at least $n * (P/100)$.*

growth: *For any two consecutive time steps t and t', where $t' = t + 1$, number of alive cells at t' is greater or equals to the number of alive cells at t.*

We can encode an instance of the SPG problem as a SAT instance. Let s_i^t be the state of the current cell i, where $1 \leq t \leq T$ and $1 \leq i \leq n$. Given a SPG problem, we encode it as a SAT formula F_{SPG} as follows

$$F_{SPG} = F_{evolution} \cup F_{safety} \cup F_{growth} \cup F_{boundary}$$

, where, $F_{evolution}, F_{safety}, F_{growth}$, and $F_{boundary}$ are defined as follows:

$$F_{evolution} : \bigwedge_{t=1}^{T} \bigwedge_{i=1}^{n} (s_i^{t+1} = (s_{i-1}^t \oplus (s_i^t \vee s_{i+1}^t)))$$

$$F_{safety} : \bigwedge_{t=1}^{T} \sum_{i=1}^{n} s_i^t \geq n * (P/100)$$

$$F_{growth} : \bigwedge_{t=1}^{T-1} \sum_{i=1}^{n} s_i^{t+1} \geq \sum_{i=1}^{n} s_i^t$$

$$F_{boundary} : \bigwedge_{t=1}^{T} \neg s_0^t \wedge \neg s_{n+1}^t$$

Over T time steps,

- $F_{evolution}$ encodes the evolution of the population of n organisms that follows Rule 30.
- F_{safety} encodes the population safety constraint.
- F_{growth} encodes the population growth constraint.
- $F_{boundary}$ encodes the assertion that left neighbor (resp. right neighbor) of the leftmost (resp. rightmost) organism (resides outside of the *boundary* of a given population) of the population is always dead (0).

F_{SPG} is SATISFIABLE, if the population can evolve upto time step T with respect to the safety and growth constraint, otherwise, it is UNSATISFIABLE.

III. Problem Modeling and Instance Generation for the SPG benchmarks

A. Problem Modeling

picat [1] is a CSP solver, which accepts a CSP problem and converts it to a SAT CNF formula, which is inturn solved by a SAT solver hosted by **picat**. Before solving the converted CNF formula, **picat** outputs the CNF formula.

To generate instances for the SPG benchmark, we use this CNF generation feature of **picat**. First, we modelled the SPG problem in a **picat** program $picat_{SPG}$. Then, for a given set of parameter values for (T, n, P), we use this $picat_{SPG}$ model to generate CNF F_{SPG} by exploiting the CNF generation functionality of **picat**.

B. Instance Generation

We have generated a set of F_{SPG} instances with the $picat_{SPG}$ by varying the parameters T and n, while setting P to a fixed value of 70. From this set of instances, we have submitted 20 instances for SAT competition-2021 (CNF file names with prefix *spg*), 10 of which are interesting[1].

References

[1] Picat, http://picat-lang.org/resources.html, Accessed: 2020-04-09

[2] Stephen Wolfram, A new kind of science. Wolfram-Media 2002, ISBN 978-1-57955-008-0, pp. I-XIV, 1-1197.

[3] Rule 30 , https://mathworld.wolfram.com/Rule30.html, Accessed: 2020-04-09

[1]Not too easy (solvable by MiniSat in a minute) or too hard (unsolvable by the participants own solver within one hour on a computer similar to the nodes of the StarExec cluster)

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Bipartite Perfect Matching Benchmarks

Cayden R. Codel, Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant
Carnegie Mellon University, Pittsburgh, United States

INTRO

The pigeonhole and mutilated chessboard problems are challenging benchmarks for most SAT solvers not employing special reasoning techniques. The solvers that do employ special techniques can efficiently solve the canonical versions of these two problems, but may fail with even slight problem variations. To evaluate and improve the robustness of SAT solvers, we designed a benchmark family of perfect matching problems on bipartite graphs that generalizes the pigeonhole and mutilated chessboard problems [1]. Our benchmark generator supports various encodings and randomized constructions. These formulas are generally hard in the absence of reasoning techniques resilient under randomness and encoding variations.

BIPARTITE PROBLEM AND ENCODING

Random bipartite graphs are used to explore non-structured problem instances of the perfect matching problem. The *density* of a bipartite graph with node partitions of size n and m is defined as the ratio of the number of edges to the number of possible edges $n \times m$. To generate a random connected bipartite graph, edges are added randomly to a random spanning tree until the desired density is reached.

Given a connected bipartite graph, a Boolean variable is associated with each edge such that a satisfying assignment is the edges in a perfect matching. The problem is encoded as a CNF with at-least-one (ALO) constraints on nodes from the larger partition and at-most-one (AMO) constraints on nodes in the smaller partition. This is the *Sparse* problem encoding. The *Full* problem encoding is derived by using both ALO and AMO constraints for each node.

AMO constraints are encoded in three ways: Pairwise, Sinz, and Linear,

Pairwise$(x_1, ..., x_n)$ is the pairwise set of binary clauses with no auxiliary variables:

$$(\overline{x}_i \vee \overline{x}_j) \text{ with } 1 \leq i < j \leq n$$

Sinz$(x_1, ..., x_n)$ introduces signal variables that propagate the AMO condition:

$$\overline{x}_i \vee s_i \quad \text{for } 1 \leq i \leq n \qquad\qquad \overline{s}_i \vee s_{i+1},\ \overline{s}_i \vee \overline{x}_{i+1} \quad \text{for } 1 \leq i < n$$

Linear$(x_1, ..., x_n)$ introduces variables to split up the Pairwise encoding when $n > 4$:

$$\text{Pairwise}(x_1, x_2, x_3, y) \wedge \text{AMO}(\overline{y}, x_4, .., x_n)$$

The **Mixed** AMO constraint option selects one of the three AMO encodings at random for each node independently. Note the signal s_n could be left out for the Sinz encoding of AMO, and is in our implementation.

BOUNDED VARIABLE ELIMINATION ON PIGEONHOLE

Experimental findings [2] revealed a performance decline for top-tier solvers when bounded variable elimination (BVE) [3] was enabled on pigeonhole formulas. To explore this phenomenon, we started with a pigeonhole formula using the Sparse problem encoding and Pairwise AMO encoding, then applied BVE to some set of variables and gave solvers the new formula to solve. We found that specific variable elimination orderings generated formulas that are difficult for all solvers tested. Namely, eliminating n variables coming from independent pigeons and independent holes. Notably, this elimination ordering is forced in the Full problem encoding for solvers that employ BVE.

BENCHMARKS

We submitted 21 benchmarks to the 2021 SAT Competition. The first 17 formulas represent three configurations for random bipartite problem generation: (1) Sparse with Pairwise AMO, (2) Sparse with Mixed AMO (denoted by MIX in the naming), and (3) Full (denoted by B in the naming) with Mixed AMO. For each configuration we construct iteratively larger graphs with partition sizes n from $15..20$, with the exception of the first configuration starting at $n = 16$. Each graph has edges added until a density of 0.5 is reached which is generally hard as seen in 1.

4 formulas are pigeonhole formulas with n from $11..14$ and the Sparse with Pairwise AMO encoding. BVE is applied to n variables for each (denoted by e# in the naming), with eliminated variables selected from independent pigeons and independent holes. This elimination order proves difficult for previous competition winners shown in 2.

All formulas are UNSAT.

REFERENCES

[1] C. Codel, J. Reeves, M. Heule, and R. Bryant, "Bipartite perfect matching benchmarks," in *Proceedings of Pragmatics of (SAT)*, 2021.

[2] J. Reeves and M. Heule, "The impact of bounded variable elimination on solving pigeonhole formulas," in *Proceedings of Pragmatics of (SAT)*, 2021.

[3] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 3569. Springer, 2005, pp. 61–75.

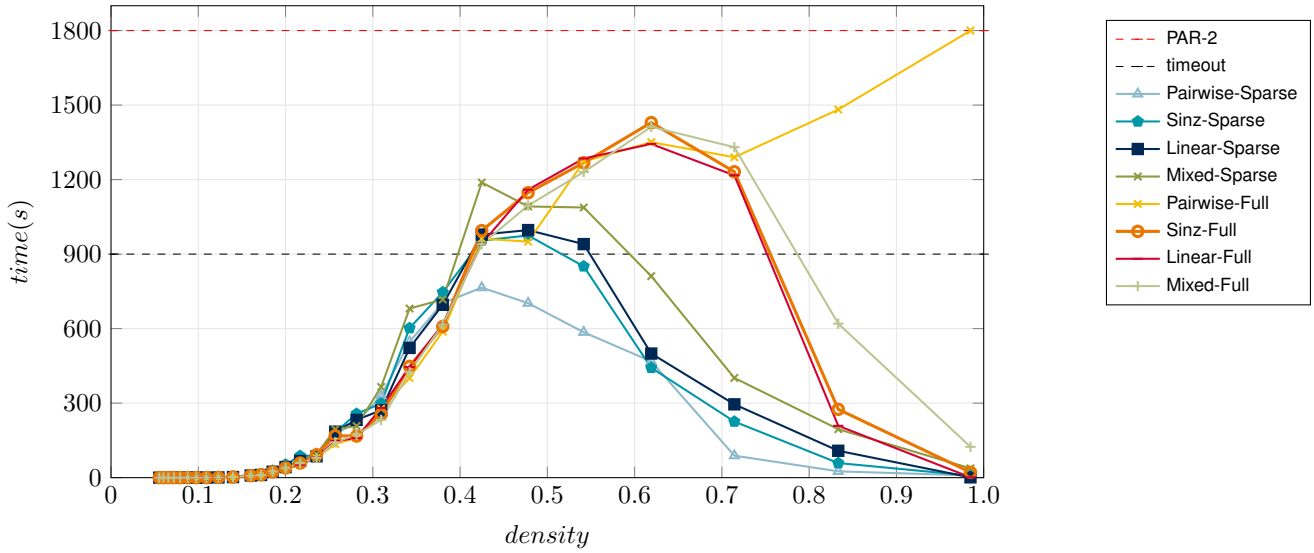


Fig. 1. Average execution time for KISSAT (2020 version) over randomly generated bipartite matching problems using a 900 second timeout and 1800 second PAR-2 score. For each density value, 60 random bipartite graphs are generated with a fixed edge count (130). Experiments cover the different AMO and problem encodings described above. Problems are harder around the 0.5 density, and the Mixed AMO encodings are difficult for the respective Sparse and Full encodings.

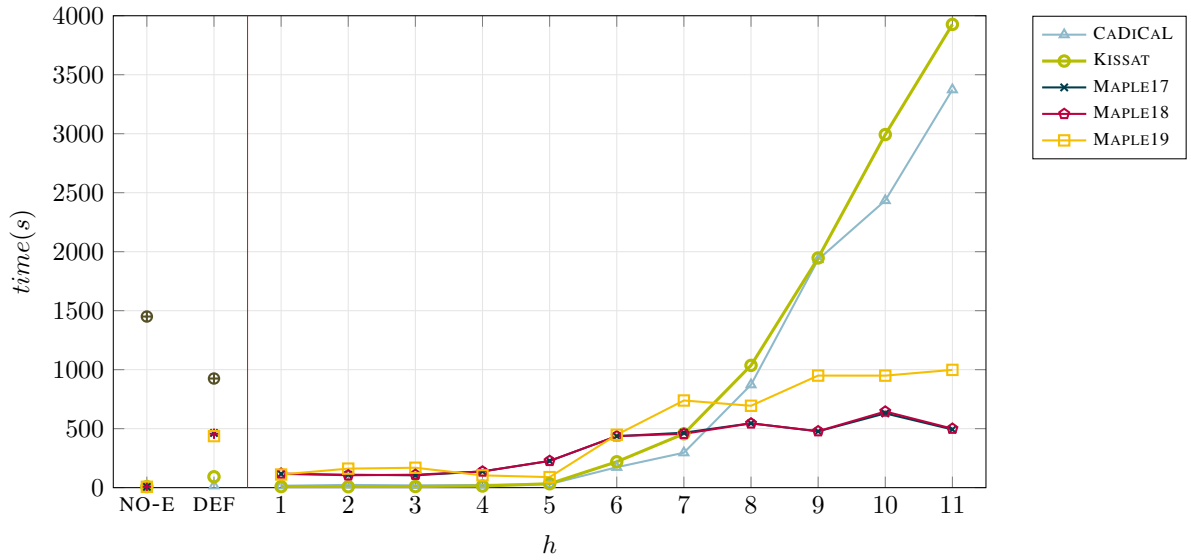


Fig. 2. Execution time on BVE instances of pigeonhole formula $n = 11$. In each h instance (x-axis) 12 variables are eliminated, selected from independent pigeons and h independent holes. NO-E is the solvers on pigeonhole $n = 11$ with BVE disabled, and DEF is the default configuration on the pigeonhole formula. Solvers are previous SAT competition winners. $h = 11$ represents the independent pigeon/hole BVE instance used in the $n = 11$ benchmark, and this variable elimination ordering is extended to $n = 12, 13, 14$. It is the most difficult formula for the solvers, though some experience larger performance degradation.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Hamiltonian Cycle Instances using the Chinese Remainder Encoding

Marijn J.H. Heule
Carnegie Mellon University, Pittsburgh, United States

Intro

Satisfiability (SAT) solvers have become very powerful tools to solve many hard combinatorial problems in a broad range of applications. However, the quality of the encoding can have a significant impact on the effectiveness of a SAT solver, in particular for problems with complicated constraints. One such problem is the Hamiltonian Cycle Problem (HCP), which has a constraint requiring that the edges form exactly one cycle.

Recently two new HCP encodings have been proposed. Both encodings assign a binary index to each vertex using $k = \lceil \log_2 |V| \rceil$ variables per vertex. The first one is based on linear-feedback shift registers (LFSR) [1], [2]. LFSR loops through the numbers $\{1, \ldots, 2^k - 1\}$ by shifting a binary number by one position to the left and puts the parity of some bits in the vacated position. This facilitates a compact SAT encoding. The second encoding uses a binary adder that loops through the numbers $\{0, \ldots, 2^k - 1\}$ in ascending order and returns to 0 after $2^k - 1$ [3]. The binary adder encoding requires auxiliary variables, more clauses, and/or longer clauses compared to LFSR. Yet, the binary adder is more effective as it facilitates quick refutation of some subcycles, e.g., cycles of odd length.

In a recent paper [4], we present the Chinese remainder encoding that aims to combine the best of the incremental SAT, binary adder, and LFSR approaches. From the incremental approach, we borrow the observation that only some subcycles need to be blocked. From the binary adder approach we borrow techniques to easily refute some subcycles. Finally, from the LFSR approach we borrow the compact encoding with short clauses without auxiliary variables.

Chinese Remainder Encoding

Given a graph $G = (V, E)$, the Chinese remainder encoding enforces that all but one cycle has length $0 \pmod m$ for a given m. The cycle that includes the initial vertex is special and is enforced to have length $|V| \pmod m$. The encoding exploits the Chinese remainder theorem to enforce these lengths: it suffices to enforce the length of cycles to be $0 \pmod{p_i}$, respectively $|V| \pmod{p_i}$, for each prime factor p_i of m. By picking m such that it has multiple small prime factors, the encoding will be compact with lots of propagation. Table I shows the results for various values of m on 8 graphs from the Flinders Hamiltonian Cycle Problem Challenge Set when solving the formulas using CaDiCaL. For small m, the encoding typically produces multiple cycles. However, for larger m (but much smaller than $|V|$), the encoding tends to produce a single cycle (thus Hamiltonian).

Benchmarks

We submitted 24 benchmarks to the 2021 SAT Competition: for each graph listed in Table I, we submitted the formula produced by $m \in \{60, 105, 420\}$. As the table shows, most of these formulas can be solved using CaDiCaL in a couple of minutes. All benchmarks are satisfiable.

TABLE I
Runtime statistics in seconds of the selected Flinders HCP challenge graphs using CaDiCaL and various values for the cycle length. The symbols ✓ and ✗ denote whether the satisfying assignment represents a single or multiple cycles, respectively.

graph #	2	6	12	60	105	420
424	9.81 ✗	665.18 ✗	340.11 ✗	307.71 ✗	494.11 ✓	**488.70** ✓
446	13.24 ✗	334.62 ✗	169.52 ✗	380.47 ✗	**573.38** ✓	722.23 ✓
470	17.08 ✗	166.16 ✗	152.31 ✗	933.36 ✗	501.91 ✗	**840.89** ✓
491	0.06 ✗	22.04 ✗	**7.47** ✓	34.45 ✓	123.36 ✓	135.22 ✓
506	0.11 ✗	31.75 ✗	**19.24** ✓	33.48 ✓	28.73 ✓	63.20 ✓
522	0.63 ✗	5.66 ✗	32.95 ✓	133.40 ✓	**30.40** ✓	67.03 ✓
526	0.05 ✗	24.16 ✗	71.67 ✓	**34.37** ✓	34.69 ✗	158.69 ✓
529	0.40 ✗	17.90 ✗	60.19 ✓	48.09 ✓	**42.33** ✓	365.58 ✓

References

[1] S. W. Golomb, *Shift Register Sequences*. Aegean Park Press, 1982.
[2] M. Haythorpe and A. Johnson, "Change ringing and Hamiltonian cycles: The search for Erin and Stedman triples," *EJGTA*, vol. 7, pp. 61–75, 2019.
[3] N.-F. Zhou, "In pursuit of an efficient SAT encoding for the Hamiltonian cycle problem," in *Principles and Practice of Constraint Programming*, H. Simonis, Ed. Cham: Springer International Publishing, 2020, pp. 585–602.
[4] M. J. H. Heule, "Chinese remainder encoding for Hamiltonian cycles," in *Theory and Applications of Satisfiability Testing – SAT 2021*, C.-M. Li and F. Manyà, Eds. Cham: Springer International Publishing, 2021, pp. 216–224.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Database Repair for Multivalued Dependencies

Sima Jamali
Simon Fraser University
Vancouver, Canada
sja88@sfu.ca

Babak Salimi
University of California San Diego
San Diego, United States
bsalimi@ucsd.edu

David Mitchell
Simon Fraser University
Vancouver, Canada
mitchell@cs.sfu.ca

***Abstract*—We describe a set of SAT solver benchmark instances (CNF formulas) obtained from the problem of finding a minimal repair for a relational database to satisfy a multivalued dependency constraint.**

***Index Terms*—MVD, SAT encoding, database repair**

I. Multivalued Dependency (MVD)

A MultiValued Dependency (MVD) for a relational database is a constraint between sets of attributes in a relation. An MVD holds when there are three attributes, say X,Y and Z, where for each value of Z there is a specific set of values for each of X and Y, but the values of X and Y are independent [1]. An MVD is a tuple-generating dependency, meaning that presence of certain tuples in the relation imply that other tuples must also be present. If a relation does not satisfy an MVD, the relation can be modified to satisfy the constraint (repaired) either by adding tuples or removing tuples. (An empty database satisfies an MVD constraint). For a database D that does not satisfy an MVD constraint, the minimal database repair problem is to find another database D^* at minimal distance from D that satisfies the MVD. As distance function we use the symmetric difference, i.e. $|D - D^*|$. Our encoding as SAT is a decision version of the MaxSAT encoding of [2].

II. SAT Encoding

If database D does not satisfy an MVD constraint ϕ, it can be repaired either by adding tuples or by removing tuples. Each CNF formula expresses, for a given database D and MVD constraint ϕ, the question: Is it possible to modify D so that it satisfies ϕ by adding at most i_1 tuples and/or removing at most i_2 tuples?

Consider database D with schema $(X, Y, W_1, \dots W_n)$, where the MVD constraint is on X, Y and some W_i. Let the (finite) domains of X and Y be $Dom(X)$ and $Dom(Y)$. We treat the remaining attributes as one variable Z, with $Dom(Z) = \prod_{W_i \in W_1 \dots W_n} Dom(W_i)$. Now define the database D* from D as:

$$D^*(\mathbf{X}_1, \mathbf{Y}_2, \mathbf{Z}) = D(\mathbf{X}_1, \mathbf{Y}_1, \mathbf{Z}) \wedge D(\mathbf{X}_2, \mathbf{Y}_2, \mathbf{Z})$$

D^* is the repaired version of database D that satisfies MVD by adding all the missing tuples.

To encode the problem, we associate to each tuple t in D^* (which includes all tuples of D) a variable V_t. Algorithm 1 generates the encoding (for the slightly simplified cases where $i_1 = i_2$). We express the MVD constraint as a set of clauses of the form $(V_{t_1} \wedge V_{t_2}) \longrightarrow V_{t_3}$, where t_1 and t_2 are tuples in D and t_3 is a tuple that must, as a consequence, be in D to satisfy ϕ. This is done by the second for loop in Algorithm 1. We also include clauses representing the fact that we are allowed to add up to i_1 clauses (those in D^* but not in D) or remove up to i_2 clauses. This is done by the first loop in Algorithm 1. It ensures that, for each subset $i_2 + 1$ existing tuples, not all can be removed, by including $\neg(\neg V_{t_1} \wedge ... \wedge \neg V_{t_{i+1}})$. A similar encoding ensures that, for each set of $i_1 + 1$ missing tuples (in D), not all can be added, by including $\neg(V_{t_1} \wedge ... \wedge V_{t_{i+1}})$.

Algorithm 1: Encodes problem of deciding D can be repaired to satisfy a MVD for with at most $i - 1$ deletions or $i - 1$ additions as a CNF formula.

Input: A database D with vairables $\mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z}$
Output: A CNF Ψ
Compute $D^*(\mathbf{X}_1, \mathbf{Y}_2, \mathbf{Z}) = D(\mathbf{X}_1, \mathbf{Y}_1, \mathbf{Z}) \wedge D(\mathbf{X}_2, \mathbf{Y}_2, \mathbf{Z})$
for $t_1, ...t_i \in D^*$ **do**
  If $t_1, ...t_i \in D$, add a clause $(V_{t_1} \vee ... \vee V_{t_i})$ to Ψ
  If $t_1, ...t_i \in D^* - D$ add a clause $(\neg V_{t_1} \vee ... \vee \neg V_{t_i})$ to Ψ
Compute
$C(\mathbf{X}_1, \mathbf{Y}_1, \mathbf{X}_2, \mathbf{Y}_2, \mathbf{Z}) = D^*(\mathbf{X}_1, \mathbf{Y}_1, \mathbf{Z}) \wedge D^*(\mathbf{X}_2, \mathbf{Y}_2, \mathbf{Z})$
for $t \in C$ **do**
  $t_1 \leftarrow t[\mathbf{X}_1, \mathbf{Y}_1, \mathbf{Z}]$; $t_2 \leftarrow t[\mathbf{X}_2, \mathbf{Y}_2, \mathbf{Z}]$;
  $t_3 \leftarrow t[\mathbf{X}_1, \mathbf{Y}_2, \mathbf{Z}]$
  Add a clause $(\neg V_{t_1} \vee \neg V_{t_2} \vee V_{t_3})$ to Ψ

III. Instance Naming

The file names are of the form MVD-*database-sequential#*-i_1-i_2. Our instances submitted to the 2021 SAT Solver competition are generated from random subsets 300 to 600 rows of the database https://archive.ics.uci.edu/ml/datasets/adult. The resulting file names then are of the form MVD-ADS-*sample#*-i_1-i_2.

Funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), through a Discovery Grant to the third author.

References

[1] S. Abiteboul, R. Hull, and V. Vianu, "Foundations of Databases,", January 1955.
[2] B. Salimi, L. Rodriguez, B. Howe and D. Suciu, "Interventional fairness: Causal database repair for algorithmic fairness," Proceedings of the 2019 International Conference on Management of Data, pp. 793-810, August 2019.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

SAT Encodings for Testing Prime and Quadratic Residue

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn

***Abstract*—Here we present SAT encodings of prime and quadratic residue testing. The quadratic residue testing problem is to ask whether there exists an integer x such that $x^2 = a \bmod p$, where a and p are integers given. If an integer p is prime, there exist no integers x and y such that $x \times y = p, x > 1, y > 1$. Although several algorithms in cryptography can test prime and quadratic residue efficiently, so far no SAT solver can solve it efficiently.**

I. Introduction

The quadratic residue testing problem is formalized as follows:

Given three positive integer a and p, find an integer x such that $x^2 = a \bmod p$.

The Jacobi symbol is a generalization of the Legendre symbol, which can be used to compute quadratic residues by the law of quadratic reciprocity and many of the properties of the Legendre symbol.

The prime testing problem is formalized as follows:

Given a positive integer p, find two integers x and y such that $x \times y = p, x > 1, y > 1$.

Prime testing can implement efficiently by sieve of Eratosthenes, sieve of Euler, Solovay-Strassen primality testing algorithm [1] and Rabin-Miller testing algorithm etc.

Here we encode two problems mentioned above into SAT problems directly. By our observation, no known SAT solver can solve efficiently the resulting SAT problems. That is, they are more difficult than the original problems.

II. Encoding Primality Testing

We translate the primality testing problem into a SAT problem by encoding directly $x \times y = p, x > 1, y > 1$. The pseudocode of this encoding algorithm is shown in Algorithm 1. In this algorithm, we assume that x and y are denoted by binary variable strings $x_m, \ldots x_2, x_1$ and $y_n, \ldots, y_2, y_1$, respectively. Every binary y_k is processed, middle result f is updated. Every update need generate new two binary variable strings $z_m, \ldots z_2, z_1$ and $f_{m+n}, \ldots f_2, f_1$. Therefore, we require at least 2mn middle binary variables.

III. Encoding Quadratic Residue Testing

Encoding quadratic residue testing is the same as encoding primality testing. It can be done by replacing y with x, since $x^2 = a$ can be viewed as a special instance of $x \times y = p$. Therefore, we can get a SAT encoding algorithm for testing quadratic residue by rewriting y and p into x and a, deleting the encoding of $x \neq 1$, $y \neq 1$ in Algorithm 1, and adding the encoding of $(f_{m+n} \ldots f_2 f_1) = (a_{m+n} \ldots a_2 a_1) \bmod p$, where a and p are constants. If the sizes are the same, the SAT problem generated by quadratic residue testing is more difficult than one generated by primality testing.

Algorithm 1 Encode $x \times y = p$

x is denoted by binary variable string $x_m, \ldots x_2, x_1$
y is denoted by binary variable string $y_n, \ldots, y_2, y_1$
p has binary expansion $(p_{m+n} \ldots p_2 p_1)_2$
middle result f is denoted by binary variable string $f_{m+n}, \ldots f_2, f_1$
encode $x \neq 1$, $y \neq 1$
for $k = 1$ to n **do**
  encode $z_m \ldots z_2 z_1 = x_m \wedge y_k \ldots x_2 \wedge y_k x_1 \wedge y_k$
  encode $f_{k+m} \ldots f_{k+2} f_{k+1} := z_m \oplus f_{k+m} \ldots z_2 \oplus f_{k+2} z_1 \oplus f_{k+1}$
end for
encode $(f_{m+n} \ldots f_2 f_1) = (p_{m+n} \ldots p_2 p_1)$

References

[1] Solovay, Robert M.; Strassen, Volker: A fast Monte-Carlo test for primality, SIAM Journal on Computing,6 (1), 84C85, 1977.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Sliding Tile Puzzles

Robert Clausecker and Benjamin Kaiser
Zuse Institute Berlin
Berlin, Germany
{clausecker,kaiser}@zib.de

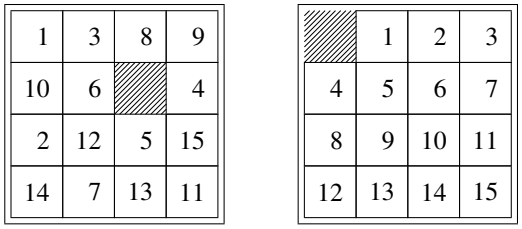


Fig. 1. a permuted and a solved *15 puzzle*

***Abstract*—We translate 5×5 sliding tile puzzle (24 puzzle) instances into CNF formulæ to compare the performance of SAT solvers with standard heuristic search methods in this domain. We find that SAT solvers still have a long way to go before they might be competitive in this problem domain.**

I. Introduction

Sliding tile puzzles comprise an $n \times m$ rectangular tray holding square tiles numbered 1 to $(nm - 1)$ with one spot empty. The objective is to permute the tiles such that their are arranged in order. To get there, the state of the puzzle can be changed by shifting tiles adjacent to the blank spot into the blank spot.

In the domain of heuristic search, sliding tile puzzles are frequently used as NP-hard model problems for graph search methods. They are particularly useful due to their simple and regular structure and admit use of many advanced search techniques.

With this submission, we would like to understand how well SAT methods might be suitable for solving this problem. While SAT solvers lack dowmain specific heuristics, they are able to attack the problem in ways that aren't really accessible to tree-search methods, e.g. by drawing conclusions from parts of the puzzle configuration that can be reused in other parts.

II. Encoding

The basic decision problem is: can a given configuration of the $(n \times n - 1)$ puzzle be transitioned into the solved configuration within k moves? This decision problem is encoded into a SAT instance by creating $k + 1$ sets of variables, each representing one puzzle configuration. Clauses are added encoding that adjacent configurations must be related by performing a single move. Furthermore, the first configuration must be equal to the problem configuration and there must exist a configuration equal to the solved configuration.

Each configuration is represented as an array of $n \times n$ vectors of literals where each vector $t_{i,j}$ represents the number of the tile at grid location (i, j). The grid is rotated horizontally and vertically such that the blank spot (numbered 0) is always in the top left corner. Two vectors of bits h and v encode in one-hot encoding where the top left tile ends up after the grid is rotated.

The move relation is encoded using two literals m_0 and m_1 encoding if the move taken was up, down, left, or right. By checking the polarity of these literals in the model found by the solver, the solution to the puzzle can be extracted.

Using these literals, we then check if the tiles on the board moved according to the move taken. We also check h and v to ensure that moves across the border do not occur.

III. Submitted Benchmarks

As a sample instance, we picked problem 50 of Korf's instances of the 24 puzzle [1]. As the full problem with its 113 step solution is too difficult to be solved by current SAT solvers, we simplified it by tracing the solution of the problem and taking the puzzle configurations obtained with distances $k = 30 \ldots 60$ to the solved configuration. With rising k, the configurations become progressively harder to solve.

For each such k, two SAT instances were generated. One instance has a move budget of k and is satisfiable. The other instance has a move budget of $k - 2$ and is unsatisfiable. This way, both the capability to solve SAT and UNSAT instances is exercised using the same type of instance.

Acknowledgment

We want to express our gratitude towards the organisers of the SAT Competition 2021 for making such an event possible. Additionally we like to thank Florian Schintke for his support and the IT and Data Services members of the Zuse Institute Berlin for providing the infrastructure.

References

[1] Richard E. Korf and Ariel Felner, "Disjoint Pattern Database Heuristics", Artificial Intelligence 134(1–2), p. 9–22, 2020.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Minimal Superpermutation SAT Benchmarks

Martin Mariusz Lester
Department of Computer Science
University of Reading
Reading, United Kingdom
m.lester@reading.ac.uk
0000-0002-2323-1771

***Abstract*—This benchmark consists of several SAT-based encodings of instances of the minimal superpermutation problem.**

I. Introduction

The minimal superpermutation problem [1] asks, for a positive integer n, what is the smallest sequence of the digits 1–n that contains every permutation of $[1, n]$ as a subsequence?

For example, for $n = 4$, the minimal superpermutation has length $l = 33$. If the first permutation in the sequence is fixed to be 1234, then it is uniquely determined to be:

$$123412314231243121342132413214321$$

For $n = 5$, $l = 153$, but the sequence is not unique [2]. For higher values of n, the length of the minimal superpermutation is not known. For example, for $n = 6$, $861 \leq l \leq 872$.

An instance of the superpermutation problem can be elegantly encoded as an instance of the Travelling Salesman Problem (TSP). The best known automated methods for solving instances of the superpermutation problem use dedicated TSP solvers. This method was used to verify minimality for $n = 5$ and find the smallest known sequence for $n = 6$ [3]. As the TSP is NP-complete, instances can be translated into SAT instances, but the resulting SAT instances are often hard for current solvers.

The instances in this benchmark suite instead use direct encodings of the superpermutation problem into SAT. The instances encode the decision problem of whether a superpermutation (or a prefix of a superpermutation) of length l for n distinct digits exists, rather than the optimisation problem of finding the smallest l for a particular n.

II. Encodings

The instances directly encode a sequence of l digits drawn from $[1, n]$ and a combinatorial circuit to recognise whether the sequence is a superpermutation.

The 1st layer of the circuit recognises individual permutations. Each permutation consists of $n!$ digits; a permutation recognising circuit checks whether the values of the digits match those expected for a particular permutation. A permutation could start at any of the l digits, except those at the end of the sequence, so roughly l copies of each permutation circuit are needed.

In the 2nd layer of the circuit, for each permutation, the outputs of each copy of the permutation recognising circuit are ORed together. The 3rd layer of the circuit ANDs together the outputs of the 2nd layer, evaluating to true only if all permutations exist in the sequence.

Different instances in the benchmark use different encodings of the digits and the circuit. There are 3 different encodings of the digits:

1) *Binary* encoding, using $\log n$ bits per digit.
2) *One-hot* encoding, using n bits per digit, with k encoded as bit k set to 1 and all other bits to 0.
3) *Unary* encoding, using n bits per digit, with k encoded using the k least significant bits set to 1 and all remaining bits set to 0.

For each encoding digit encoding, there are 2 variants:

1) A *non-strict* encoding, where a digit's bits are constrained only by the permutation recognising circuit.
2) A *strict* encoding, where extra clauses constrain a digit's bits only to valid encodings of a digit; this sometimes allow a smaller encoding of the permutation recognising circuit.

The 1st layer of the circuit has 2 variants:

1) *Flat:* The permutation recognising circuit is a large AND over equality of all digits.
2) *Tree:* The permutation recognising circuit is built from a tree of permutation prefix recognising circuits. Where two permutations share a common prefix, they share circuitry to recognise that prefix.

In total, this amounts to $3 \cdot 2 \cdot 2 = 12$ different encodings.

III. Instances

The benchmark suite contains instances of 3 slightly different problems:

1) Find the minimal superpermutation for $n = 4$ with $l = 33$. These instances are easy, with MiniSAT 2.2.1 solving them in less than 1 minute.
2) Show that the minimal superpermutation for $n = 4$ with $l = 33$ is unique, once the first permutation is fixed. These instances add a clause to instances from the preceding set that forbids the known superpermutation, making them unsatisfiable. These instances are still relatively easy, with MiniSAT solving the hardest in just over 2 minutes.
3) Find a prefix of a superpermutation for $n = 5$ with either $l = 21$ and $g = 15$ permutations, or $l = 26$ and $g = 19$

permutations. These instances are harder, with MiniSAT solving 7 out of the 12 $l = 21$ instances in under 10 minutes and none of the $l = 26$ instances.

In the final set of instances, the check for g permutations was encoded by converting the SAT instance to a Pseudo-boolean (PB) instance, where it could easily be added as a cardinality constraint. Then the cardinality constraint was converted back to SAT using `pbencoder` from `pblib` [4].

The values of l for the final set of instances were chosen to be hard but plausible within the 5000 second time limit used in the SAT Competition. With the PB formulation, the default configuration of the PB solver *clasp* [5] was able to solve 8 out of 12 of the $l = 26$ instances within this time limit.

All timings are for an Intel i5-7500 CPU running at 3.40GHz.

REFERENCES

[1] D. A. Ashlock and J. Tillotson, "Construction of small superpermutations and minimal injective superstrings," in *Conference on Algebraic Aspects of Combinatorics and Sundance Conference and International Conference on Algol 68 Implementation*, ser. Congressus Numerantium, no. v. 93. Utilitas Mathematica Pub. Incorporated, 1993, pp. 91—98. [Online]. Available: https://books.google.co.uk/books?id=f5PgAAAAMAAJ

[2] N. Johnston, "Non-uniqueness of minimal superpermutations," *Discret. Math.*, vol. 313, no. 14, pp. 1553–1557, 2013. [Online]. Available: https://doi.org/10.1016/j.disc.2013.03.024

[3] R. Houston, "Tackling the minimal superpermutation problem," *CoRR*, vol. abs/1408.5108, 2014. [Online]. Available: http://arxiv.org/abs/1408.5108

[4] T. Philipp and P. Steinke, "Pblib – a library for encoding pseudo-boolean constraints into cnf," in *Theory and Applications of Satisfiability Testing – SAT 2015*, ser. Lecture Notes in Computer Science, M. Heule and S. Weaver, Eds. Springer International Publishing, 2015, vol. 9340, pp. 9–16.

[5] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "*clasp* : A conflict-driven answer set solver," in *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, ser. Lecture Notes in Computer Science, C. Baral, G. Brewka, and J. S. Schlipf, Eds., vol. 4483. Springer, 2007, pp. 260–265. [Online]. Available: https://doi.org/10.1007/978-3-540-72200-7_23

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

At Least Two Solutions

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

***Abstract*—This document describes the AtLeastTwo tool, as well as the benchmarks that have been submitted to the SAT competition 2021. The tool takes a CNF and encodes a new CNF which is satisfiable only if the initial CNF has at least two satisfying solutions. The difference of the two solutions can be restricted to input variables only, to support formulas that have been encoded with auxiliary variables.**

I. Introduction

When encoding problems, one might ask whether given problem has more than one solution. This property is for example useful when generating logic puzzles to know whether a generated puzzle has multiple solutions. A related property is to encode that a formula has exactly one solution. Similarly, the encoding could be extended to ask for at least two satisfying assignments, or at most k satisfying assignments. However, the result formula would grow quadratically with k. The current tool only supports asking for at least two solutions. Other extensions would be to encode that at least a given number of K assignments need to be different. In this case, the currently used *at-least-k* constraint – a clause – would have to be replaced by a properly encoded cardinality constraint.

II. Encoding At Least Two Solutions

The input for the AtLeastTwo tool is an uncompressed CNF file with the formulas F. This formula has n variables, and m clauses, where we assume all variables n to be present. As in CNF, variables are represented as integers, we will use math rules to describe transformations. The resulting formula is basically encoded by the following steps: (1) duplicate F into a formula G, but add the offset n to all variables to obtain fresh variables, (2) encode the equivalence for all variable pairs in F and G, respecting the offset, and (3) enforce that are least one equivalence from (2) is falsified. The three steps will be explained in more details next.

A. Duplicate Input Formula

As a first step, we duplicate all clauses in F. During this step, for all clauses C_i, we add the offset n to the variable representation of all literals l_i when creating the new clauses.

$$G := \bigwedge_{C_i \in F} \bigvee_{l_i \in C_i} l_i + n$$

B. Encode Solution Equality

To encode whether two variable assignments are equivalent, we need to introduce a new variable for each existing variable in F, or at least for the range of selected variables whose satisfying assignment should be different. Let e be the number of variables from 1 to e which should have at least one different assignment. As the default case, the number of variables to consider is the number of variables in the input, i.e. $e = n$.

$$E := \bigwedge_{i=1}^{e} a_i \leftrightarrow (f_i \leftrightarrow g_i)$$

,

where a_i are variables that do not occur in F nor in G. Note, the variables f_i and g_i again have the offset of n again, i.e. $g_i = f_i + n$.

C. Force Inqeuality of Solutions

When combining all formulas above, we obtain a formula that encodes the same formulas twice, and a sub-formula that indicates whether the assignments for both formulas are equal:

$$R' = F \wedge G \wedge E$$

.

A model for this formula can assign the same truth value for all variable pairs in F and G. To make sure, the models are different, we need to ensure that at least one pair of variables is not equal, by encoding an at-least-one constraint for the new auxiliary variables:

$$R = F \wedge G \wedge E \wedge \bigvee_{i=1}^{e} \neg a_i$$

.

Formula R is the resulting formula that will be generated by the tool.

D. Properties of the Generated Formula

First, we discuss the relationship between the satisfiability of the input formula F and the produced formula R. When the input formula F is unsatisfiable, the resulting formula R is also unsatisfiable.

Next, when F has exactly one model, the truth values of this model can be assigned to satisfy F. Formula G will can only be satisfied with the same set of assignments. Consequently, all variables a_i will be assigned to $\top$. In this case, the last subformula of R will be falsified.

In the final case, multiple model I and J exist to satisfy F. Then, I can satisfy F, and J can satisfy G by respecting the offset n. As I and J are diferent, at least one variable a_i will be assigned to $\bot$, resulting in a satisfying assignment for R. Note, such a satisfying assignment to R contains the two models I and J that satisfy the input formula F.

III. The Submitted Benchmark

The submitted benchmark formulas have been generated by using formulas from previous SAT competition benchmarks. The initial set of formulas to select the submitted formulas from is the robust benchmark for tuning SAT solvers from [1].

IV. Availability

The source of the tool is publicly available under the MIT license at https://github.com/conp-solutions/cnfmiter.

References

[1] H. H. Hoos, B. Kaufmann, T. Schaub, and M. Schneider, "Robust benchmark set selection for boolean constraint solvers," in *Revised Selected Papers of the 7th International Conference on Learning and Intelligent Optimization - Volume 7997*, ser. LION 7. Berlin, Heidelberg: Springer-Verlag, 2013, p. 138–152.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

A Naive SAT-Encoding of Cluster Editing

Stefan Mengel
CNRS, UMR 8188, Centre de Recherche en Informatique de Lens (CRIL), Lens, F-62300, France
Univ. Artois, UMR 8188, Lens, F-62300, France

INTRO

A *cluster graph* is defined to be a graph whose connected components are all cliques. The *cluster editing problem* is, given a graph G, to turn it into a cluster graph by applying as few edge modifications, i.e. edge additions and deletions, as possible. This problem is well known to be NP-hard.

Due to numerous applications, cluster editing has been studied extensively, in particular in the parameterized algorithms community, see e.g. [1], [2] for an overview. There is also quite an extensive literature on practical approaches to cluster editing, see again [2] for a discussion and [3]. Note that practical implementations largely rely on integer linear programming or specialized branch & bound methods. SAT-encodings have been tried, but were largely inefficient [4]. Since cluster editing is the problem treated in PACE 2021, the yearly Parameterized Algorithms and Computational Experiments Challenge [5], we thought it might be interesting to see how state-of-the-art SAT solvers fare on the problem. To this end, we provide several (naively encoded) instances for the SAT competition 2021.

DESCRIBING THE ENCODING

It is easy to see that a graph G is a cluster graph if and only if for every triple u, v, w of vertices in G the following is true: if uv and uw are edges in G, then vw is an edge of G as well. This observation is the basis of our encoding. Given a vertex set V, for ever potential edge uv, we add an indicator variable x_{uv} that is true if and only if uv is an edge in the graph we want to describe. Then for every triple $u, v, w \in V$, we add the three clauses

$$\bar{x}_{uv} \vee \bar{x}_{uw} \vee x_{vw}, \bar{x}_{uv} \vee \bar{x}_{vw} \vee x_{uw}, \bar{x}_{uw} \vee \bar{x}_{vw} \vee x_{uv}.$$

By what we said before, the satisfying assignments of the resulting formula F_V are exactly the encodings of all cluster graphs on the vertex set V.

Now given a graph G on vertex set V and an integer k, we can encode the question of if there is a cluster graph G' that we can get from G by modifying at most k edges as follows: let a_G be the assignment of the variables x_{uv} that corresponds to the graph G. Then we add a single cardinality constraint c that forces that at most k variables may differ from the value they get in the assignment a_G.

BENCHMARKS

The above gives for every graph G and a bound k an encoding. We implemented this using PySAT [6], in particular for the convenient generation of cardinality constraints. As input graphs, we used public instances of the exact track of PACE 2021 [4], which in turn were taken from different public sources [7], [8], [9], [10], [11], [12] or randomly generated; for more details see also the description of the benchmarks by the organizers of PACE 2021 that will be published soon. While the size of our encoding is cubic in the number of vertices of the graph, due to the limited size of the graphs in the benchmark set, we can still mostly generate the CNF-formulas reasonably quickly.

To have interesting bounds for k that are close to the optimal value, we ran a simple greedy heuristic that tries to turn an input into a cluster graph by modifying edges uv that contribute to many conflicting triples u, v, w. For not too big graphs, this tends to give values that are reasonably close to the optimal value. To generate unsatisfiable instances (or get closer to the optimum), we subtract a small number for some SAT instances. When k is close to the optimum, the resulting instances are surprisingly hard in our preliminary tests, even for some small graphs. In particular, this also tends to be true for satisfiable instances.

We submitted 20 benchmarks to the 2021 SAT Competition, 6 unsatisfiable, 14 satisfiable. The naming format of the instances is `edit_distancexxx_kk.cnf` where `xxx` corresponds to the number of the graph in the PACE 2021 exact track benchmark set and `kk` is the bound that is tested. Note that for some graphs we have submitted instances for several bounds.

OUTLOOK

As stated above, the instances that we created are, at least in our preliminary tests, hard to solve even for small graphs. We have tried another encoding of cluster editing that was inspired by coloring problems, where we essentially gave every node a color enconding the clique it is in in the modified graph. This encoding was smaller, but the solvers performed even worse on it.

For our encoding, there are different ways in which it can be improved. For example, we have performed tests in which we used known preprocessing rules for cluster editing to add unit clauses to the encoding. Unfortunately, this made only very little difference for the solver runtime, so we have not included it in the submitted instances.

It would be interesting to see if there are better SAT-encodings of cluster editing that make SAT-solvers competitive with MIP-solvers and branch & bound techniques.

References

[1] V. Froese, "Fine-grained complexity analysis of some combinatorial data science problems," Ph.D. dissertation, Technical University of Berlin, Germany, 2018. [Online]. Available: https://nbn-resolving.org/urn:nbn:de:101:1-201810170200455373521 4

[2] S. Böcker and J. Baumbach, "Cluster editing," in *The Nature of Computation. Logic, Algorithms, Applications - 9th Conference on Computability in Europe, CiE 2013, Milan, Italy, July 1-5, 2013. Proceedings*, ser. Lecture Notes in Computer Science, P. Bonizzoni, V. Brattka, and B. Löwe, Eds., vol. 7921. Springer, 2013, pp. 33–44. [Online]. Available: https://doi.org/10.1007/978-3-642-39053-1_5

[3] S. Hartung and H. H. Hoos, "Programming by optimisation meets parameterised algorithmics: A case study for cluster editing," in *Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers*, ser. Lecture Notes in Computer Science, C. Dhaenens, L. Jourdan, and M. Marmion, Eds., vol. 8994. Springer, 2015, pp. 43–58. [Online]. Available: https://doi.org/10.1007/978-3-319-19084-6_5

[4] A. Nichterlein, personal communication.

[5] Pace 2021: Call for participation. [Online]. Available: https://pacechallenge.org/2020/10/22/PACE-2021-CFP/

[6] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, 2018, pp. 428–437. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_26

[7] Wcs data archives. [Online]. Available: http://www.icsi.berkeley.edu/wcs/data.html

[8] The 20 newsgroups text dataset. [Online]. Available: https://scikit-learn.org/0.19/datasets/twenty_newsgroups.html

[9] Transclust example data. [Online]. Available: https://transclust.compbio.sdu.dk/online_service/web.php

[10] S. Rahmann, T. Wittkop, J. Baumbach, M. Martin, A. Truss, and S. Böcker, "Exact and Heuristic Algorithms for Weighted Cluster Editing," in *Proc. of Computational Systems Bioinformatics (CSB 2007)*, vol. 6, 2007, pp. 391–401.

[11] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[12] S. Böcker, S. Briesemeister, Q. B. A. Bui, and A. Truß, "A fixed-parameter approach for weighted cluster editing," in *Proceedings of the 6th Asia-Pacific Bioinformatics Conference, APBC 2008, 14-17 January 2008, Kyoto, Japan*, ser. Advances in Bioinformatics and Computational Biology, A. Brazma, S. Miyano, and T. Akutsu, Eds., vol. 6. Imperial College Press, 2008, pp. 211–220. [Online]. Available: http://www.comp.nus.edu.sg/%7Ewongls/psZ/apbc2008/apbc050a.pdf

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Verifying String Safety Properties in AWS C99 Package with CBMC

Muhammad Osama and Anton Wijs
Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

***Abstract*—In this paper, state-of-the-art proofs are generated with harness using the CBMC bounded model checker for the Amazon Web Services C99 core package. We check, in particular, the safety properties of the *String compare* routine with various *loop unwinding* settings. The generated proof has proven to be reasonably hard to solve using modern SAT solvers. It has many variable-clause redundancies which are not only challenging for a SAT solver but also useful to assess the performance of different simplification techniques.**

I. Introduction

Bounded Model Checking (BMC) [1], [2] determines whether a model M satisfies a certain property φ expressed in temporal logic, by translating the model checking problem to a propositional satisfiability (SAT) problem or a Satisfiability Modulo Theories (SMT) problem. The term *bounded* refers to the fact that the BMC procedure searches for a counterexample to the property, i.e., an execution trace, which is bounded in length by an integer k. If no counterexample up to this length exists, k can be increased and BMC can be applied again. This process can continue until a counterexample has been found, a user-defined threshold has been reached, or it can be concluded (via k-induction [2]) that increasing k further will not result in finding a counterexample. CBMC [3], [4] is an example of a successful BMC model checker that uses SAT solving. CBMC can check ANSI-C programs. The verification is performed by *unwinding* the loops in the program under verification a finite number of times, and checking whether the bounded executions of the program satisfy a particular safety property [5]. These properties may address common program errors, such as null-pointer exceptions and array out-of-bound accesses, and user-provided assertions.

II. Benchmarks

In this paper, we are interested in verifying the safety properties of the *compare* routine implemented in the String data structure of the Amazon Web Services (AWS) C99 core package. The proof covers the following:

- Memory allocation failure and access violations
- Pointer/floating-point overflow
- Data types conversion

We generated 41 different formulas using a *loop unwinding* upper-bound in the range $[600, 1000]$, with an increasing step of 10. These bounds make the SAT formulas achieve 100% coverage of all functionalities. All problems are written in this format:

`string_compare_safety_cbmc_unwinding_<x>`

where `x` denotes the unwinding value. The first and the last formulas are solved via MiniSat [6] within 470 and 3000 seconds respectively on a machine with Intel Core i5-7600 operating at 3.5 GHz. The solving time of the rest of the benchmarks are expected to be monotonically increasing.

III. Acknowledgment

We would like to thank Daniel Kroening and Natasha Jebbo for referring us to the AWS C99 package and helping with the configuration of the proof environment.

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*. Springer, 1999, pp. 193–207.
[2] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.
[3] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.
[4] D. Kroening and M. Tautschnig, "CBMC – C Bounded Model Checker," in *TACAS*. Springer, 2014, pp. 389–391.
[5] D. Kroening and O. Strichman, *Decision Procedures*. Springer, 2016.
[6] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *SAT*, ser. LNCS, vol. 2919. Springer, 2004, pp. 502–518.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

PEQNP Python Library Benchmarks

1st Oscar Riveros
PEQNP
Santiago, Chile
oscar.riveros@peqnp.science

***Abstract*—The formulas that are generated by PEQNP Library represent some particular instances of the following Problems: Sum of 3 Cubes [1] and Maximum Constrained Partition [2].**

I. Introduction

The PEQNP System its an automatic CNF encoder and SAT Solver for General Constrained Diophantine Equations and NP-Complete Problems, fully integrated with Python [3].

II. SAT COMPETITION 2021 BENCHMARKS

The collected formulas have generated with PEQNP Library for the following problems:

A. Sum of 3 Cubes

Let be $x, y, z \in \mathbb{Z}$ and $x^3 + y^3 + z^3 = t$ search for x, y, z. with $t \in \{87, 96, 91, 80, 39, 84, 75, 30, 52, 74\}$.

B. Maximum Constrained Partition

Given a finite set S of $2n$ elements in $\mathbb{N}$ decide if exist a partition of $X \cup Y = S$ with $X \cap Y = \emptyset$, $|X| = |Y|$ and $\Sigma X = \Sigma Y$.

III. Instances

```
maximum_constrained_partition_10_bits_n200.cnf
maximum_constrained_partition_11_bits_n200.cnf
maximum_constrained_partition_12_bits_n200.cnf
maximum_constrained_partition_13_bits_n200.cnf
maximum_constrained_partition_14_bits_n200.cnf
maximum_constrained_partition_15_bits_n200.cnf
maximum_constrained_partition_16_bits_n200.cnf
maximum_constrained_partition_17_bits_n200.cnf
maximum_constrained_partition_18_bits_n200.cnf
maximum_constrained_partition_19_bits_n200.cnf
sum_of_3_cubes_37_bits_87.cnf
sum_of_3_cubes_42_bits_96.cnf
sum_of_3_cubes_50_bits_91.cnf
sum_of_3_cubes_51_bits_80.cnf
sum_of_3_cubes_52_bits_39.cnf
sum_of_3_cubes_76_bits_84.cnf
sum_of_3_cubes_87_bits_75.cnf
sum_of_3_cubes_94_bits_30.cnf
sum_of_3_cubes_108_bits_52.cnf
sum_of_3_cubes_145_bits_74.cnf
```

Thanks to all supporters of http://www.peqnp.com projects.

References

[1] A compendium of NP optimization problems, Nikos Drakos, Ross Moore. http://www.csc.kth.se/ viggo/wwwcompendium/node152.html
[2] Which integers can be expressed as a sum of three cubes in infinitely many ways?, Mathoverflow. https://mathoverflow.net/questions/138886/which-integers-can-be-expressed-as-a-sum-of-three-cubes-in-infinitely-many-ways
[3] PEQNP Mathematical Solver, Oscar Riveros, http://www.peqnp.com

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Multiplier Input Decomposition Instances generated by ToughSAT

1st Shunyang Bi, 1st Zhang Qu, 5th Hailong You
School of Microelectronics
XiDian University
Xi'an China
shybi@stu.xidian.edu.cn,
quzhang2019@stu.xidian.edu.cn,
hlyou@mail.xidian.edu.cn

2nd Meihua Liu
School of Electronic and Computer Engineering
Peking University Shenzhen Graduate School
Shenzhen Guangdong China
liumh@pku.edu.cn

3rd Pengfei Li, 4th Yang Zhang
EDA Group
SMIT Holdings Limited
Shenzhen Guangdong China
379401663@qq.com,
yanzhang@smit.com.cn

***Abstract*—this description introduce our instances to the SAT Competition 2021. We generated instances that would select proper input decomposition from multiplication of large numbers.**

I. DATA

In the circuit design of n-digit multiplication multiplier, using the ordinary multiplication algorithm needs n^2 times of multiplication, while $3 * n^{\log_2 3}$ ($3 * n^{1.585}$) times of multiplication in the fast multiplication algorithm(Karatsuba's algorithm). For example, let x and y be represented as n-bit strings in a cardinality b. For any positive integer less than n, two given numbers can be written as:

$$x = b^m * x_1 + x_0$$

$$y = b^m * y_1 + y_0$$

Where x and y are less than b^m, that is to say:

$$x * y = (b^m * x_1 + x_0) * (b^m * y_1 + y_0)$$

Let

$$z_0 = x_1 * y_1$$

$$z_1 = x_0 * y_1 + x_1 * y_0$$

$$z_2 = x_0 * y_0$$

Then,

$$x * y = b^{2m} * z_0 + b^m * z_1 + z_2$$

In this process, it takes 4 times multiplication operations to decompose the multiplication. But in fast multiplication algorithm, z_1 can be expressed as:

$$z_1 = (x_1 + x_0) * (y_1 + y_0) - x_1 * y_1 - x_0 * y_0$$

And we just need 3 times of multiplication. In the actual circuit, we need to verify whether this decomposition method is feasible.

II. SELECTION

Whether the input of a designed multiplier circuit can be decomposed into multiplication factor based on fast multiplication algorithm is very important for our circuit design. The multiplier constraint is defined as the multiplier inputs of the circuit we designed. These inputs have appeared in our circuit design. We define the input in the multiplier as f_1, f_2, and assign them according to the actual design circuit. TABLE Ⅰ shows the running time of 20 instances in Minisat.

TABLE I. RESULTS WITH MINISAT FOR 20 INSTANCES SUBMITTED FOR SAT COMPETITION-2021.

Instance name	f_1	f_2	*Minisat Time*	*Status*
Circuit_multiplier_18.cnf	71472475	35478902	5000	UNKNOWN
Circuit_multiplier_20.cnf	17783402	274475	206.98	SAT
Circuit_multiplier_22.cnf	47545134	8348021	1659.06	SAT
Circuit_multiplier_23.cnf	54513144	34802174	595.69	SAT
Circuit_multiplier_24.cnf	479613144	1802174	5000	UNKNOWN
Circuit_multiplier_25.cnf	96131440	802174	5000	UNKNOWN
Circuit_multiplier_26.cnf	61314404	2174734	5000	UNKNOWN
Circuit_multiplier_28.cnf	144024741	773457	1444.49	SAT
Circuit_multiplier_29.cnf	77340057	40247415	5000	UNKNOWN
Circuit_multiplier_33.cnf	979147121	175171	253.31	SAT
Circuit_multiplier_34.cnf	59147121	7325171	3073.5	SAT
Circuit_multiplier_35.cnf	98325171	1441721	4539.31	SAT
Circuit_multiplier_36.cnf	179325171	93411721	5000	UNKNOWN
Circuit_multiplier_37.cnf	9263325171	721721	181.99	SAT
Circuit_multiplier_47.cnf	977317491	7894567	5000	UNKNOWN
Circuit_multiplier_48.cnf	435678915	9647851	4307.78	SAT
Circuit_multiplier_45.cnf	169117141	16773165	703.21	SAT
Circuit_multiplier_17.cnf	8642475	6547892	500.02	SAT
Circuit_multiplier_53.cnf	92147042	13795646	5000	UNKNOWN
Circuit_multiplier_54.cnf	92776646	85247042	5000	UNKNOWN

III. TOOLS

We used ToughSAT [1] to assist in adding the constraints of multiplier and generating the CNF formulas.

REFERENCES

[1] Joseph Bebel, "Harder SA T Instances from Factoring with Karatsuba and Espresso,"in Proceedings of SAT Competition 2019. [Online]. Available: https://helda.helsinki.fi/handle/10138/306988

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Computing Preferred Extensions for Abstract Argumentation

Xindi Zhang, Shaowei Cai*

[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{zhangxd,caisw,chenzh}@ios.ac.cn

***Abstract*—In this document, we describe how to generate SAT instances though a general SAT-based abstract argumentation solver called MiniAF. We only focus on solving the reasoning tasks of preferred semantic on the ICCMA benchmarks.**

I. Introduction

SAT-based method is one of the most popular formal argumentation approaches for solving reasoning tasks of the abstract argumentation framework [1]. In this document, we use a general solver called MiniAF [6] to solve tasks from International Competition on Computational Models of Argumentation(ICCMA). For clearer understanding, we repeating the encoding method in this document.

II. Basic concepts

A given abstract argumentation framework(AF) $AF = (A, R)$ can be represented by a directed graph, where A is a set of arguments and $R \subseteq A \times A$ is the relation. For two arguments $a, b \in A$, the relation aRb means that a attacks b (which can be represented by $a \to b$ as well), and we denote $a^- = \{b|bRa\}$. A set $S \subseteq A$ defends an argument $b \in A$ if for all a with aRb there is $c \in S$ with cRa. Semantic σ represents a kind of property over a set of arguments, and a σ-extension $E \subseteq A$ is a argument set with the property σ. There are definitions of some important semantic extensions.

- An extension E is $conflict\text{-}free$ (cf) iif there are no arguments $a, b \in E$ with aRb;
- An extension E is $admissible$ (adm) iif E is cf and E defends every $a \in E$;
- An extension E is $complete$ (co) iif E is adm and if E defends a then $a \in E$;
- An extension E is $preferred$ (prf) iif E is maximal co.

Given a semantic $\sigma \in \{co, prf\}$ and an AF $AF = (A, R)$, an argument $a \in A$ is $skeptically\ accepted$ in AF if a is contained in every σ-extensions, is $credulously\ accepted$ in AF if a is contained in some σ-extensions. There are some tasks base on a given AF $AF = (A, R)$ and an argument a.

- EE-σ: Enumerate all extensions $E \subseteq A$ that are σ-extensions;
- SE-σ: Return an extension $E \subseteq A$ that is a σ-extension;
- DC-σ: Decide if a is credulously accepted under σ;
- DS-σ: Decide if a is skeptically accepted under σ.

This work was supported by Beijing Academy of Artificial Intelligence (BAAI), and Youth Innovation Promotion Association, Chinese Academy of Sciences [No. 2017150].

* Corresponding author

III. Labelling Encoding Method

This section introduces an equivalent way to define different types of semantics by labelling encoding method [4], [6]. Given a set of arguments A, a $labelling$ L mapping each argument $a \in A$ to $\{in, out, undec\}$, which means that a is accepted, rejected or the status is undecided, respectively. The set of all $labellings$ for a given $AF = (A, R)$ is denoted as $\zeta(AF)$.

$L \in \zeta(AF)$ is called a $complete\ labelling$ (co-L) iif for any $a \in A$ holds:

- $L(a) = in \Leftrightarrow \forall b \in a^-, L(b) = out$;
- $L(a) = out \Leftrightarrow \exists b \in a^-, L(b) = in$.

A co-L $L \in \zeta(AF)$ is equal to a prf-extension iif L maximize the set of arguments labelled in.

IV. Encoding for Complete Semantics

This section gives the classic encoding method for complete semantics [2] used in MiniAF [6]. Given an AF $AF = (A, R)$ with $|A| = k$, and $\Phi : \{1, ..., k\} \to A$ is an indexing bijection.

At first, we define three symbols I_i, O_i, U_i for each argument $a \in AF$ with indexing i, and for each argument $a \in AF$, a can labelled exact one type label.

$$\bigwedge_{i \in \{i,\dots,k\}} ((I_i \vee O_i \vee U_i) \wedge (\neg I_i \vee \neg O_i) \wedge (\neg I_i \vee \neg U_i) \wedge (\neg O_i \vee \neg U_i)) \tag{1}$$

By the definition, for each argument $a \in AF$ without any attackers, a should be labelled in.

$$\bigwedge_{\{i|\Phi(i)^- = \emptyset\}} (I_i \wedge O_i \wedge U_i) \tag{2}$$

Then, for each argument $a \in AF$ with at least one attacker: $L(a) = in \Rightarrow \forall b \in a^-, L(b) = out$; $L(a) = in \Leftarrow \forall b \in a^-, L(b) = out$.

$$\bigwedge_{\{i|\Phi(i)^- \neq \emptyset\}} \Big(\bigwedge_{\{j|\Phi(j) \to \Phi(i)\}} \neg I_i \vee O_j \Big) \tag{3}$$

$$\bigwedge_{\{i|\Phi(i)^- \neq \emptyset\}} \Big(I_i \vee \Big(\bigvee_{\{j|\Phi(j) \to \Phi(i)\}} \neg O_j \Big) \Big) \tag{4}$$

At last, for each argument $a \in AF$ with at least one attacker: $L(a) = out \Rightarrow \exists b \in a^-, L(b) = in$; $L(a) = out \Leftarrow \exists b \in a^-, L(b) = in$.

$$\bigwedge_{\{i|\Phi(i)^- \neq \emptyset\}} \Big(\neg O_i \vee \Big(\bigvee_{\{j|\Phi(j) \to \Phi(i)\}} \neg I_j \Big)\Big) \tag{5}$$

$$\bigwedge_{\{i|\Phi(i)^- \neq \emptyset\}} \Big(\bigwedge_{\{j|\Phi(j) \to \Phi(i)\}} I_j \vee O_i \Big) \tag{6}$$

All the above formulas (1)-(6) make up a conjunctive normal form (CNF) Π, which can be solved by a given SAT solver. At last, to enumerate all extensions, MiniAF excluding previous model s by add a formula $\neg s$ to Pi after each time a model is found by the SAT solver, until the SAT solver return that there are no more model (UNSAT).

V. Preferred Semantics and Related Tasks

MiniAF uses an improved PrefSAT algorithm [3] for computing $preferred\ labellings$ (prf-L). The algorithm iterates over a set of co-Ls to identify the preferred ones and optimizes the process by set inclusion to maximise co-Ls.

To decide the credulous acceptance of an argument a, the CNF Π is updated to $\Pi \wedge I_{\Phi^{-1}(a)}$. To check the skeptically acceptance of an argument a, MiniAF subsequently enumerates all prf-Ls util it finds a labelling with $L(a) \neq in$.

VI. Benchmark selection

We use MiniAF to solve the tasks of EE-prf, DS-prf and DC-prf on the benchmarks from ICCMA-17, ICCMA-19 which can be downloaded from http://argumentationcompetition.org/. Following the definition of 'interesting instance' that one should not be solved by MiniSat [5] in a minute and should be solved by our own solver within 1 hour, We select some interesting instances from the intermediate results of MiniAF, which are in the format of "$.cnf$".

References

[1] A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial intelligence*, 93(1-2):63–101, 1997.

[2] F. Cerutti, P. E. Dunne, M. Giacomin, and M. Vallati. Computing preferred extensions in abstract argumentation: A sat-based approach. In *International Workshop on Theorie and Applications of Formal Argumentation*, pages 176–193, 2013.

[3] F. Cerutti, M. Vallati, and M. Giacomin. An efficient java-based solver for abstract argumentation frameworks: jargsemsat. *International Journal on Artificial Intelligence Tools*, 26(02):1750002, 2017.

[4] G. Charwat, W. Dvořák, S. A. Gaggl, J. P. Wallner, and S. Woltran. Methods for solving reasoning problems in abstract argumentation–a survey. *Artificial intelligence*, 220:28–63, 2015.

[5] N. Eén and N. Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518, 2003.

[6] J. Klein and M. Thimm. Revisiting sat techniques for abstract argumentation. *Computational Models of Argument: Proceedings of COMMA 2020*, 326:251, 2020.

Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, University of Helsinki 2021.

Mycielski principle formulas with PR clauses

Emre Yolcu and Marijn Heule
Carnegie Mellon University, Pittsburgh, PA 15213, USA

Mycielski graphs

The *Mycielskian* [1] $\mu(G)$ of a graph $G = (V, E)$ is constructed as follows:

1) Include G in $\mu(G)$ as a subgraph.
2) For each $v_i \in V$, add a vertex $u_i \in U$ adjacent to all of $N_G(v_i)$.
3) Add a vertex w adjacent to all of U.

If G is triangle-free then so is $\mu(G)$, and $\mu(G)$ has chromatic number one higher than G.

Let $M_2 = K_2$ (the complete graph on 2 vertices). *Mycielski graphs* are an infinite family $\{M_2, \mu(M_2), \mu(\mu(M_2)), \dots\}$ of triangle-free graphs with arbitrarily high chromatic number. In particular, for each $k \geq 2$, the graph M_k has chromatic number k. Figure 1 shows the first few Mycielski graphs.

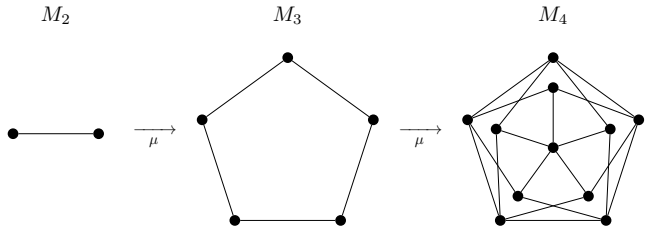


Fig. 1. The first few graphs in the family of Mycielski graphs.

Mycielski principle formulas

Given graph G and a number $k \in \mathbb{N}$, it is straightforward to encode graph coloring as a CNF formula. Specifically, the following collection of clauses, denoted $\mathrm{Color}(G, k)$, is satisfiable if and only if G is k-colorable:

$$\bigvee_{c \in [k]} x_c \qquad \text{for each } x \in V$$

$$\overline{x_c} \vee \overline{y_c} \qquad \text{for each } xy \in E,\ c \in [k]$$

The *Mycielski principle* is the falsehood stating that M_k is colorable with $k - 1$ colors. For each $k \geq 2$, the unsatisfiable formula $\mathrm{Myc}_k \coloneqq \mathrm{Color}(M_k, k - 1)$ is a CNF encoding of the Mycielski principle. The benchmarks described herein are extensions of Myc_k with satisfiability-preserving inferences in some recently introduced propositional proof systems.

Satisfiability-preserving inferences

Inferences allowed in most commonly studied propositional proof systems respect logical implication. In such proof systems, we can derive a clause F from a set Γ of clauses only if Γ logically implies F (every assignment satisfying Γ satisfies F). Alternatively, one might consider inferences that only respect satisfiability, where we can derive F from Γ only if $\Gamma \cup F$ is *equisatisfiable* to Γ (the set $\Gamma \cup F$ is satisfiable if and only if Γ is satisfiable). Examples of such inferences are blocked clause introduction [2] or the more general propagation redundant clause introduction [3]. Proof systems based on these inferences have been shown to admit short proofs of many "hard" principles [3], [4] without requiring new variables (unlike, say, Extended Frege).

Benchmarks

In recent work [5], we proved the existence of short proofs of the Mycielski principle in the propagation redundancy proof system without requiring new variables. The submitted benchmarks contain two subsets containing versions of the formulas Myc_{10} and Myc_{11} extended with parts of the proof in [5]. All of the submitted instances are unsatisfiable, and they have quasilinear-length resolution refutations. For the following description we follow the notation of [5].

$\mathtt{BC}_k$ is a set of binary clauses for Myc_k that enforce the coloring to be a well-defined function, stating that each vertex has to be assigned at most one color. $\mathtt{PR}_k$ is a set of ternary clauses for $\mathrm{Myc}_k \cup \mathtt{BC}_k$ stating that (under a condition) we can assume without loss of generality that the colors of the vertices in an *inner* layer of the Mycielski construction are the same as that of the corresponding vertices in the *outer* layer (see the second arrow in Figure 1 where the layers become apparent). $\mathtt{R1}_k$ and $\mathtt{R2}_k$ are reverse unit propagation inferences for $\mathrm{Myc}_k \cup \mathtt{BC}_k \cup \mathtt{PR}_k$ that essentially allow us to copy the edges from an outer layer in the Mycielski construction to an inner layer. The first subset of benchmark instances contain $\mathrm{Myc}_{10} \cup \mathtt{BC}_{10} \cup \mathtt{PR}_{10}$ as a core, with individual instances additionally containing $50, 60, \dots, 100$ percent of the $\mathtt{R1}_{10}$ clauses. The second subset of instances contain $\mathrm{Myc}_{11} \cup \mathtt{BC}_{11} \cup \mathtt{PR}_{11} \cup \mathtt{R1}_{11}$ as a core, with individual instances additionally containing $10, 20, \dots, 40$ percent of the $\mathtt{R2}_{11}$ clauses.

References

[1] J. Mycielski, "Sur le coloriage des graphs," *Colloquium Mathematicae*, vol. 3, no. 2, pp. 161–162, 1955.
[2] O. Kullmann, "On a generalization of extended resolution," *Discrete Applied Mathematics*, vol. 96-97, pp. 149–176, 1999.
[3] M. J. H. Heule, B. Kiesl, and A. Biere, "Strong extension-free proof systems," *Journal of Automated Reasoning*, vol. 64, no. 3, pp. 533–554, 2020.
[4] S. Buss and N. Thapen, "DRAT and propagation redundancy proofs without new variables," *Logical Methods in Computer Science*, vol. 17, no. 2, 2021.
[5] E. Yolcu, X. Wu, and M. J. H. Heule, "Mycielski graphs and PR proofs," in *Theory and Applications of Satisfiability Testing (SAT)*, 2020, pp. 201–217.

Solver Index

Benchmark Index

Author Index